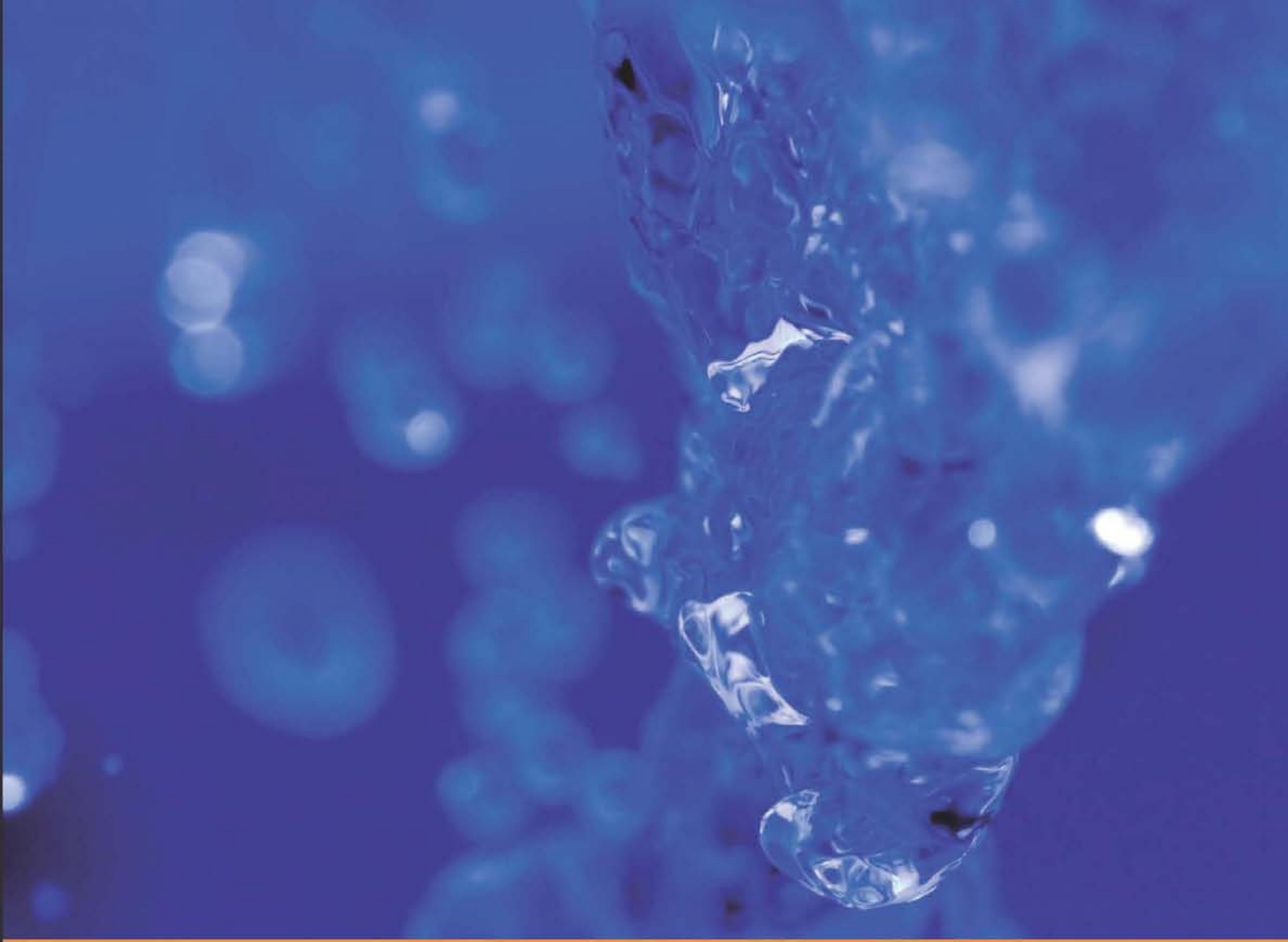


From Technologies to Solutions

Flash with Drupal

Build dynamic, content-rich Flash CS3 and CS4 applications for Drupal 6

Travis Tidwell

PACKT PUBLISHING

Flash with Drupal

Build dynamic, content-rich Flash CS3 and CS4
applications for Drupal 6

Travis Tidwell

PACKT
PUBLISHING

BIRMINGHAM - MUMBAI

Flash with Drupal

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers or distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2009

Production Reference: 1200509

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847197-58-0

www.packtpub.com

Cover Image by Gabriela y la pintura (linaza100@hotmail.com)

Credits

Author
Travis Tidwell

Reviewer
Steve Zeidner

Acquisition Editor
David Barnes

Development Editor
Swapna V. Verlekar

Technical Editor
Aditi Srivastava

Copy Editor
Ajay Shanker

Indexer
Hemangini Bari

Editorial Team Leader
Abhijeet Deobhakta

Project Team Leader
Lata Basantani

Project Coordinator
Joel Goveya

Proofreader
Laura Booth

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the author

Travis Tidwell is the founder and CTO for TMT Digital (`http://www.tmtdigital.com`), a company that specializes in the development of Flash applications for the Drupal Content Management System. He is also the sole developer for the Dash Media Player (`http://www.tmtdigital.com/project/dash_player`, `http://www.drupal.org/project/dashplayer`), which is a media player built specifically for Drupal. As well as contributing to this media player, Travis is also the author and co-maintainer for the FlashVideo module (`http://www.drupal.org/project/flashvideo`), which is a complete video solution for Drupal.

Travis graduated with a degree of Bachelors of Science in Electrical and Computer Engineering from Oklahoma State University and has worked as an Embedded Systems Engineer for companies specializing in automotive and agricultural GPS products. Travis then fell in love with web development and more specifically with Drupal and Flash, where he has developed numerous sites including `http://www.delicioso.com` for Food Networks, Ingrid Hoffmann.

In his spare time (which is rare these days), Travis enjoys the performing arts where he sings, plays the guitar, and even tap dances (go to `http://www.youtube.com` and search for "Soul Man Tap" to see him in action).

Travis currently lives in Des Moines, IA with his beautiful wife Erin, and is the proud parent of a feisty one-year-old named Brycen.

> I would like to thank my wife, Erin, who has stood by me and supported me through the long evenings and weekends where I pursued my passions, including writing this book. Without her support, I would not be where I am today.

About the reviewer

Steve Zeidner is a web developer who has been using Perl, PHP, ActionScript, Flash, and other technologies to design and code web sites and web applications since 1999. In the past, he has been involved in educational projects such as Moodle and WeBWorK while pursuing his degree in Computer Science and Engineering at The Ohio State University.

After completing his undergraduate Computer Science studies, Steve went on to develop his SQL database, web programming, and IT administration skills at the Ohio Farm Bureau Federation. In 2007, he became the lead web developer at Guardian Enterprise Studios, where he broadened his knowledge and use of web design elements with content management systems as well as PHP, Perl, and MySQL database integration. He currently resides in Columbus, Ohio and has an interest in social media and how the Web is changing the way people communicate.

Table of Contents

Chapter 8: The Hybrid Approach

Preface

This book is an in-depth discussion and tutorial session on how to integrate Flash applications with the Drupal CMS. It describes the best techniques and practices for integrating Flash technology with the power and flexibility of Drupal—by building real-life Flash applications.

In this book, you will learn how to build Flash applications that show text from within Drupal and also present images, music, and video within a single Flash application. You will also be able to take advantage of the expandable fields and content filtering provided from the CCK and Views module to add flexibility and power to your Flash applications. Finally, you will learn how to add your own custom functionality to Drupal and then utilize that from within your Flash applications, leaving you with a world of possibilities.

This book starts out as a simple introduction to Flash and Drupal technology, where you will create a simple Flash application and then embed that within Drupal. From there, each subsequent chapter will build onto the previous chapter and you will tackle new and challenging tasks. For each new task, you will take a step-by-step approach to building a real-life application that utilizes the features introduced within that chapter. You will also explore alternative design approaches that eliminate the current design challenges that developers face when building Flash-driven Drupal sites; and all this while staying true to the object-oriented principles that govern the foundation of the ActionScript 3 language. By the end of this book, you will be able to apply all the lessons learned from this book to any other use case you may encounter.

What this book covers

Chapter 1 sets the stage for the reader to learn how Flash and Drupal can combine to create a dynamic, content-rich experience for our users. We will learn how to embed Flash applications within Drupal, as well as learn about some important modules that make it easier to work with Flash in Drupal.

Chapter 2 builds a "Hello World" application. We will say hello to the world in Flash using Drupal-driven content. However, unlike any other "Hello World" tutorial, we will learn the important concepts of asynchronous programming and how that relates to working with Drupal content in Flash.

Chapter 3 covers how to add custom content to our Flash applications using the popular **Content Construction Kit** (**CCK**). We will illustrate this concept by building a hypothetical Recipe Flash application designed for a Drupal cooking web site.

Chapter 4 shows us how to use Drupal managed images to give our application a little visual flare as a visually stunning Flash application would not be complete without the integration of images. We will build on from the previous chapter by adding an image to our Recipe Flash application.

Chapter 5 explores how to use Drupal to manage a list of audio nodes and also builds a Flash application to play that music. When it comes to multimedia, Flash is the portal of choice for playing audio on a web sites.

Chapter 6 expands our custom media player to not only play music, but also show Flash videos managed from our Drupal web site, which is built onto the concept from the previous chapter. In addition, we will learn some important concepts of object-oriented practices while we reuse common components to build a media player for Drupal.

Chapter 7 explains the basics of how to take an existing Flash application and break apart the components for remote communication. We achieve this by first abstracting out separate functionalities into two separate components, and then laying the foundation for a communication gateway between the two different components. This is an essential first step to create a robust and easily maintained system, where Flash applications can be separated on a Drupal web site, thus implementing a hybrid Flash integration approach.

Chapter 8 creates the necessary components required to implement the hybrid approach. This chapter focuses on creating the bridge between two different Flash applications. Once we create this bridge, we will have the ability to control our media from a remote Flash application. In other words, we will be building a remote control for our media player that can be placed anywhere on the page, separate from the media player.

Chapter 9 builds a media player whose playlist is driven from the power of the Drupal Views module. Arguably, the most important aspect of any content-rich web site is its ability to build lists of each piece of content in a meaningful manner. The Drupal Views module gives the administrator the ability to manage the contents of their site in a meaningful list of content to present to the users. By combining this power within Flash, we can learn how to create a playlist of video nodes for our custom media player.

Chapter 10 shows how to utilize user management within a Flash application by building a User Login Flash widget. One of the most important aspects of the Drupal CMS is its ability to manage its users and protect the content of that site using a permission-based role system.

Chapter 11 shows how to add content to our Drupal web site while at the same time keep our data safe from malicious software. Not only can Flash be used to show Drupal content, but it can also be utilized to add and manipulate Drupal content from a remote Flash application.

Chapter 12 will sum up all lessons learned in this book by building a five-star voting mechanism in Flash. We will learn how to build a custom Voting Service as well as create our very own Flash driven five-star voter compatible with the popular FiveStar module.

What you need for this book

We need to install Drupal version 6, Flash CS3 or CS4, and Apache-MySQL-PHP (AMP) for this book.

Who this book is for

This book is written for developers who wish to build dynamic Flash applications. Although, we will be using Drupal for our Content Management System, the lessons learned within this book can easily be applied to other content management systems such as Joomla or WordPress. Because of this, you are not required to be familiar with Drupal. Any interaction with Drupal will be described in full detail so that anyone can follow along. As for Flash, it is not necessary to be familiar with how to use Flash since that too will be covered in this book. However, it is recommended that you have some modest understanding of ActionScript and PHP since there are many code examples in this book.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can start this off by opening up our `main.as` file and then, shift our focus to the `onNodeLoad` function."

A block of code will be set as follows:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be shown in bold:

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp"
var sessionId:String = "";
var nodeId:Number = 5;
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, and mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note via the **SUGGEST A TITLE** form on `www.packtpub.com`, or send an email to `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit `http://www.packtpub.com/files/code/7580_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately, so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of this book, and we will do our best to address it.

1
Flash with Drupal

It is no secret that the web sites that become the most popular on the Web are those which possess a series of traits that appeal to the masses; most prominently, content and beauty. Because of this trend, many web developers strive to produce sites that are not only beautiful, but also rich in content. In order to satisfy the beauty requirement, Flash has easily risen to the top as being an ideal portal to deliver dynamic and beautiful web experiences to its users. At the same time, with the explosion of content on the Internet, Drupal has stood out, among other content management systems, as a powerful and expandable means to manage content. It only seems natural, then, to combine these two incredible technologies together to create the ultimate user experience.

In this book, we will learn how to integrate Flash with Drupal by taking a hands-on approach to building real-life Flash applications for Drupal CMS. Each chapter will introduce a different and more difficult challenge, where we will continually build onto the skills learned from the previous chapters. It is my goal, that by the end of this book, you should be able to venture off on your own and build your very own Flash applications that integrate beautifully with Drupal. But before we get ahead of ourselves, we need to take a step back and understand the motivation for this book. We also need to touch base on some of the basics for integrating Flash with Drupal, which include:

- Why Flash with Drupal?
- Who is this book for?
- Getting started with Drupal
- Adding content to Drupal
- Getting started with Flash
- Creating a Flash application
- Adding Flash content to Drupal

Why Flash with Drupal?

Within the past couple of years, there has been a major paradigm shift in the world of product development. Everything from phones, web sites, and automobiles has been affected from this movement, where the importance of functionality has seemed to have been replaced with style and usability. No longer do the products that offer maximum features sell more than their competitors. Instead, the products that offer style and ease of use are the ones that rule the trade and are considered to be the "next big thing".

Web development is no stranger to this trend. Just take a look at some of your favorite web sites and you will realize how important style and usability are to the world of web development. With this movement taking hold, it is easy to see how Flash technology has risen to the top as being the portal of choice for many web site user interfaces. Flash offers many tools that make it easy to create stylish and easy-to-use applications, and because of this, many people know a Flash web site when they see one (and usually, remember it). What is not so apparent about Flash, though, is its lack of ability to effectively manage and deliver dynamic content. Fortunately, this is where Drupal shines.

With the explosion of content on the Internet, **Content Management Systems** (**CMS**) have become mainstream for any web site administrator, who wishes to manage the onslaught of new content on his/her site. Although there are many different flavors of **CMS**, Drupal is rapidly becoming the system of choice because it offers a powerful and extensible framework that can mould to any application. However, when it comes to style and usability, Drupal requires a lot of work to get the look and feel in the way that you and your visitors would expect from any top-notch web site. With that said, one can easily see how the combination of Flash and Drupal is a match made in heaven between beauty and the brain.

Who is this book for?

This book is written for developers who wish to build dynamic Flash applications, whose content is governed from a **CMS**. Although we will be using Drupal for our **CMS**, the lessons learned within this book can easily be applied to other content management systems such as Joomla or WordPress. Because of this, it is not assumed that you are familiar with how to use Drupal, and any interaction with Drupal will be described in full detail so that anyone can follow along. As for Flash, it is not necessary to be familiar with how to use Flash since the basics will also be covered in detail within this book. However, it is recommended that you have a modest understanding of ActionScript 3.0 and PHP, since there will be many code samples in this book. With that said, let's get started!

Getting started with Drupal

We will kick things off by first exploring Drupal and seeing how to utilize a few of its many features for our Flash applications. Although this will not be a complete guide to Drupal, it will give us a quick introduction, so that we can comfortably integrate its power into the Flash applications that we will create in this book. And, it all begins with the installation!

Installing Drupal

Since Drupal is a web-based application, our first goal is to install a server that will be able to run and execute the PHP code that makes up this incredible **CMS**. The most typical setup for running Drupal is to use an Apache web server along with PHP and MySQL services enabled.

Installing Apache-MySQL-PHP (AMP)

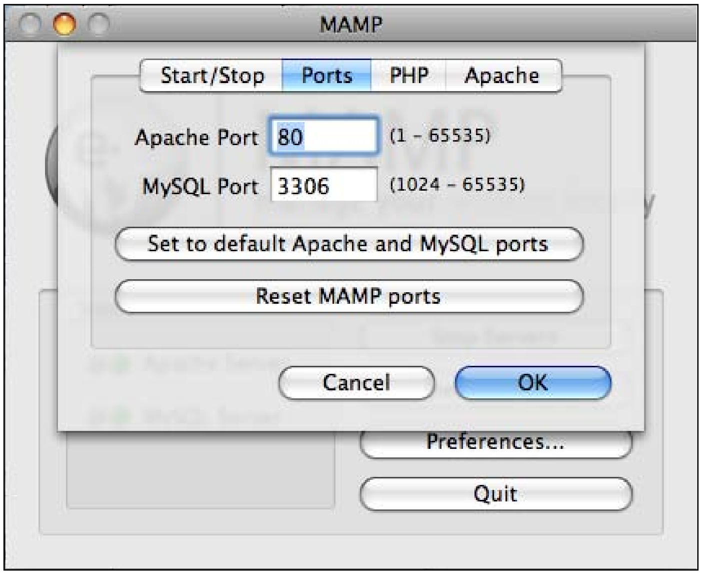
There are many ways to install Apache, MySQL, and PHP, but I would highly recommend installing a pre-built AMP package onto your computer, and then running Drupal on your machine through a local server. Luckily, there are several pre-built installers that make this step as simple as possible. Each operating system has its own version, and they all can be found at the following locations:

- Windows and Linux—XAMPP (`http://www.apachefriends.org`)
- Mac—MAMP (`http://www.mamp.info/en/index.php`)

Each of these packages has an easy-to-follow installer that will install an AMP server on your local machine, so that we can then install Drupal. After we are done with the installation of these packages, we should be able to go to the following locations within your web browser, to see a welcome page.

- Windows and Linux—`http://localhost`
- Mac—`http://localhost:8888/MAMP`

If you are using MAMP, then I would highly recommend setting the default Apache and MySQL ports by clicking on the **Preferences** button in the MAMP application. Once you are in the **Preferences** section, we will then select the **Ports** tag and click on **Set to default Apache and MySQL ports**.

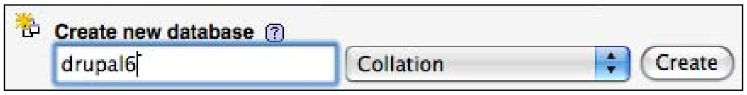


Apache Port 80 (1 - 65535)
MySQL Port 3306 (1024 - 65535)

This will make it such that you can type `http://localhost` within your browser without the port (:8888), which is consistent with the rest of this book. Now, with a web server installed, our next step is to install Drupal.

Creating the Drupal database

Once we have our AMP server running, our next task is to create a database that we will use for our Drupal installation. This simply requires running **phpMyAdmin** that comes with the AMP packages, and can usually be found by navigating to the welcome screen at `http://localhost` (for XAMPP) and `http://localhost/MAMP` (for MAMP), and then clicking on the link that says **phpMyAdmin**. Once we are inside the **phpMyAdmin** front page, we can easily create a new database called **drupal6** using the **Create new database** input field and clicking on the **Create** button.

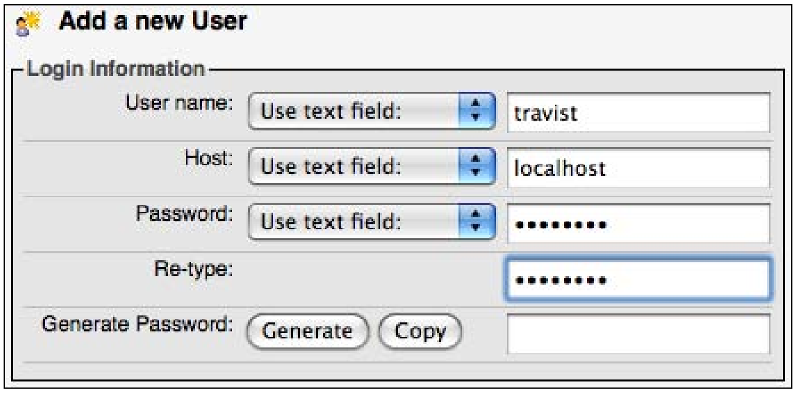


Create new database: drupal6

Now that we have a new database, we will need to create a new user who can use our Drupal database.

Creating a database user

We will create a new user by first navigating back to the main **phpMyAdmin** screen, and then clicking on the link that says **Privileges**. Once we are in the **Privileges** section, we can create a new user by clicking on the **Add a new user** link. We will then fill out the user information by providing the following information (of course, you can use your own username):

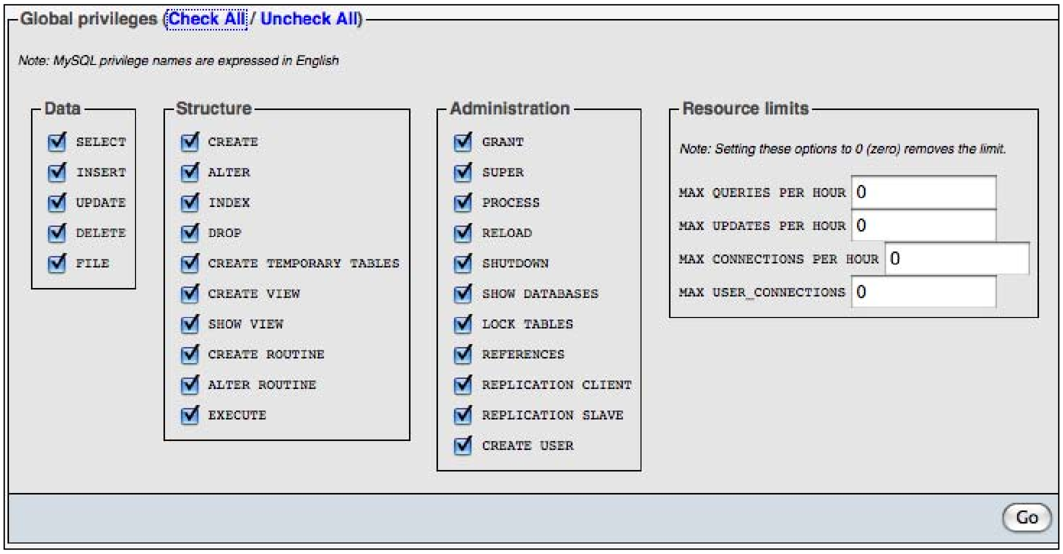


Add a new User

Login Information

User name:	Use text field:	travist
Host:	Use text field:	localhost
Password:	Use text field:	••••••••
Re-type:		••••••••
Generate Password:	Generate / Copy	

Our next task is to make sure that we give our user the correct privileges to use the database that we just created. Since we are just using this as a local server, it is completely fine to give your new user global privileges by clicking on the **Check All** link that is next to the **Global privileges** section.



Global privileges (Check All / Uncheck All)

Note: MySQL privilege names are expressed in English

Data
- ☑ SELECT
- ☑ INSERT
- ☑ UPDATE
- ☑ DELETE
- ☑ FILE

Structure
- ☑ CREATE
- ☑ ALTER
- ☑ INDEX
- ☑ DROP
- ☑ CREATE TEMPORARY TABLES
- ☑ CREATE VIEW
- ☑ SHOW VIEW
- ☑ CREATE ROUTINE
- ☑ ALTER ROUTINE
- ☑ EXECUTE

Administration
- ☑ GRANT
- ☑ SUPER
- ☑ PROCESS
- ☑ RELOAD
- ☑ SHUTDOWN
- ☑ SHOW DATABASES
- ☑ LOCK TABLES
- ☑ REFERENCES
- ☑ REPLICATION CLIENT
- ☑ REPLICATION SLAVE
- ☑ CREATE USER

Resource limits

Note: Setting these options to 0 (zero) removes the limit.

MAX QUERIES PER HOUR	0
MAX UPDATES PER HOUR	0
MAX CONNECTIONS PER HOUR	0
MAX USER_CONNECTIONS	0

We can now click on the **Go** button, at the bottom right of the **Privileges** section, to create our new user. Our next step is to increase our PHP memory, so that Drupal does not timeout when it is installing.

Increasing PHP memory

This next step is almost always overlooked when installing Drupal. By default, the AMP packages do not allocate enough PHP memory required to install Drupal. Because of this, we will need to edit the php.ini file and manually increase this value. The php.ini file can be found within the conf/php5 directory of our AMP package installation directory. Once we open the php.ini file, we will perform a search for memory_limit and change it to 32M. While we are in this file, we should also probably increase the execution and input time as follows:

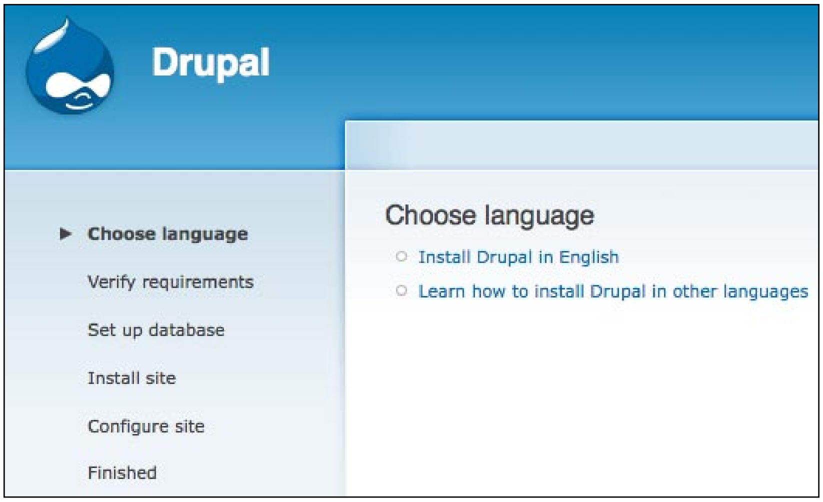
```
max_execution_time = 1000
max_input_time = 1000
memory_limit = 32M
```

After we save our changes, it is very important that we reset Apache so that these changes take effect. We can do this by opening up our AMP control application, and choosing to reset the Apache server, or we can simply close and restart this application to perform the required Apache restart. With this done, we are now ready to install Drupal!

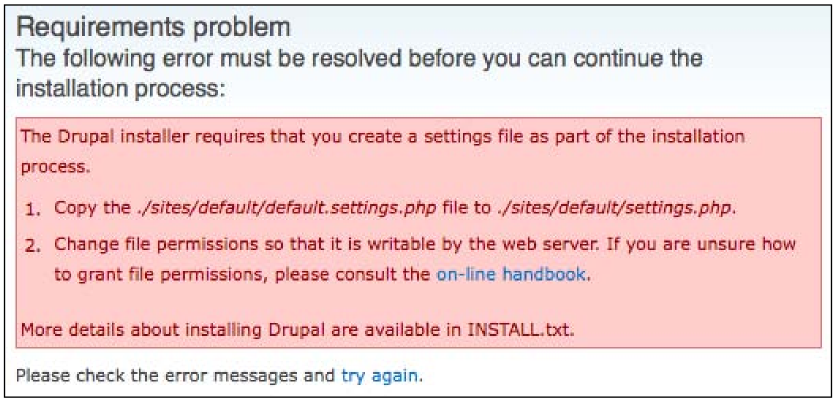
Installing Drupal

Moving right along, we can download the latest version of Drupal by going to http://www.drupal.org and clicking on the download links found on the homepage. For this book, we will be using **Drupal 6**, so select the latest version of Drupal that begins with 6. At the time of writing, the release was version 6.10, which can be downloaded at http://ftp.drupal.org/files/projects/drupal-6.10.tar.gz.

Once we have downloaded this package, the next step is to extract the contents into the document root of the web server. This is typically within the htdocs folder of the XAMPP or MAMP installation. Whatever resides in this folder is now visible when you navigate to http://localhost from your browser. For example, if we created an HTML file called index.html inside the htdocs folder, and then navigated to http://localhost, we would see that HTML file rendered within our browser as a web page. In this book, we will simply create a new folder within our htdocs called drupal6, and then extract the contents of the Drupal 6 package inside that new folder. Since we have placed all of our contents inside the htdocs/drupal6 folder, we can now open up our browser and type http://localhost/drupal6 to see the following:



We can begin our installation by clicking on the link that says **Install Drupal in English**, where we will then be greeted with the following page:



Requirements problem

The following error must be resolved before you can continue the installation process:

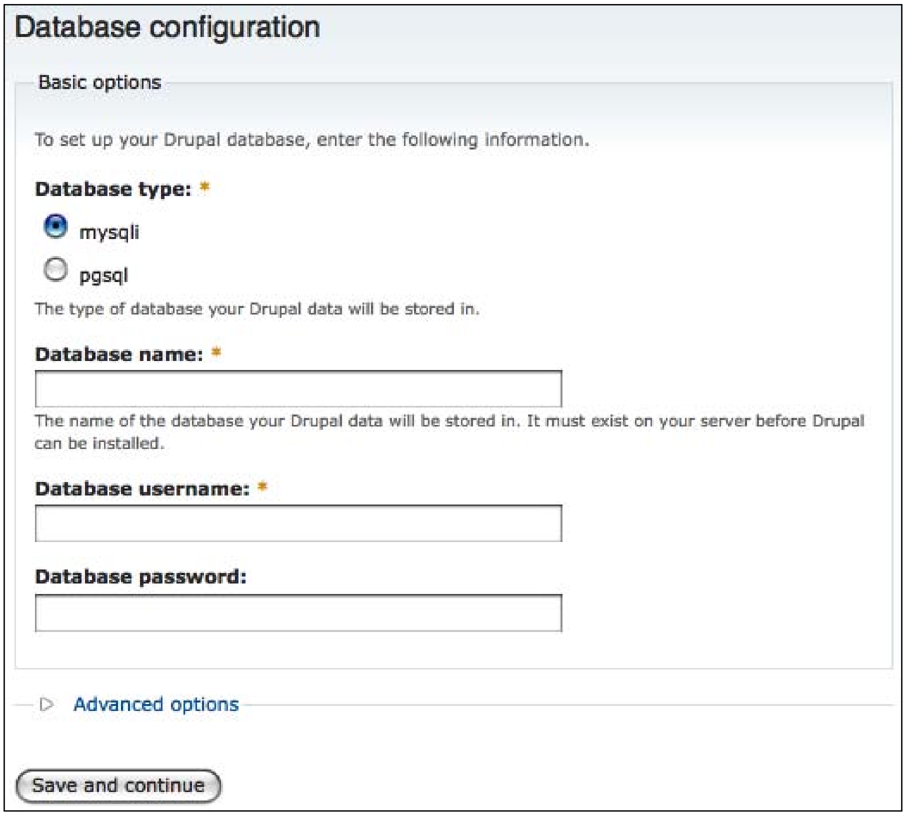
The Drupal installer requires that you create a settings file as part of the installation process.

1. Copy the *./sites/default/default.settings.php* file to *./sites/default/settings.php*.
2. Change file permissions so that it is writable by the web server. If you are unsure how to grant file permissions, please consult the on-line handbook.

More details about installing Drupal are available in INSTALL.txt.

Please check the error messages and try again.

Don't let the red scare you. What this is telling us to do is find the `default.settings.php` file within the `sites/default` folder of our extracted Drupal folder. Once we find this file, we simply need to rename it as `settings.php`. After it has been renamed, we need to change the permissions of the file so that it is writable. When we are done, we can click on the **try again** link, and see the following:



Database configuration

Basic options

To set up your Drupal database, enter the following information.

Database type: *

● mysqli

○ pgsql

The type of database your Drupal data will be stored in.

Database name: *

The name of the database your Drupal data will be stored in. It must exist on your server before Drupal can be installed.

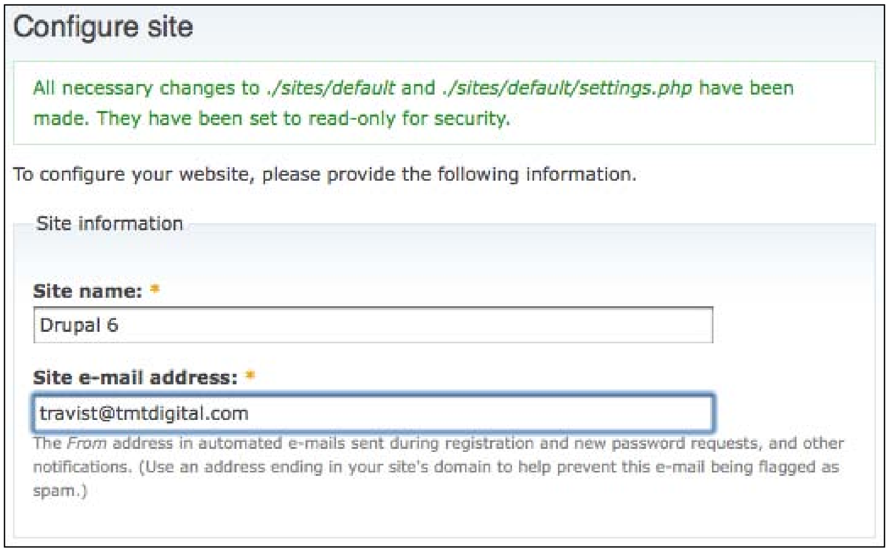
Database username: *

Database password:

From here, we will just enter all the database information that we set up in the previous section, and then click on the **Save and continue** button. This should then walk through the installation process. If it does not, then we can manually edit the `settings.php` file and manually put in our database information by changing the following line:

```
$db_url = 'mysql://travist:mypassword@localhost/drupal6';
```

We can now navigate to `http://localhost/drupal6/install.php` and kick off the installation. If all goes well, we should see the following page:



Configure site

All necessary changes to *./sites/default* and *./sites/default/settings.php* have been made. They have been set to read-only for security.

To configure your website, please provide the following information.

Site information

Site name: *

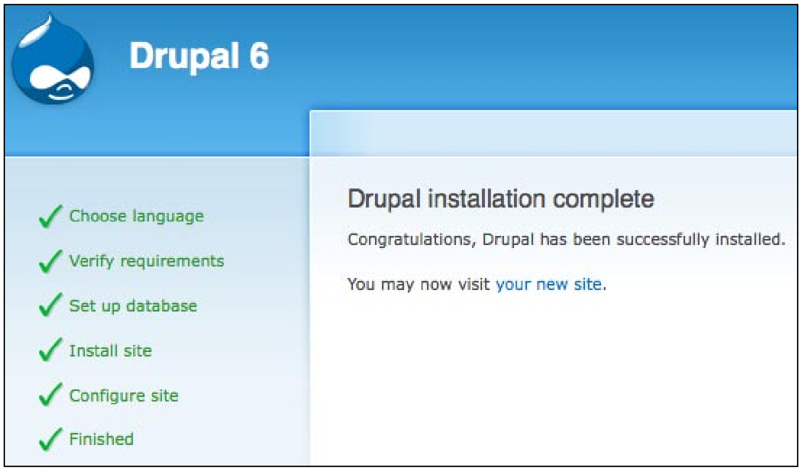
Drupal 6

Site e-mail address: *

travist@tmtdigital.com

The *From* address in automated e-mails sent during registration and new password requests, and other notifications. (Use an address ending in your site's domain to help prevent this e-mail being flagged as spam.)

At this point, we can fill out all of the initial configurations for our Drupal web site, including the **site name**, **email**, **user name**, **password**, and so on. When we are done, we can click on the **Save and Continue** button at the bottom of the page, where we should see the following:

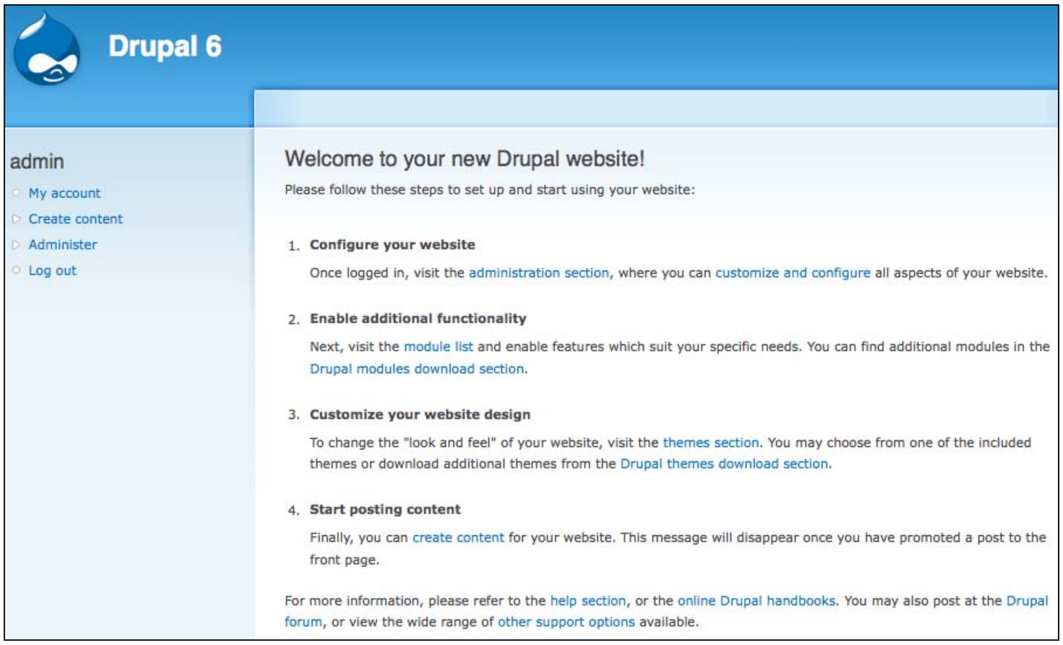


Drupal installation complete

Congratulations, Drupal has been successfully installed.

You may now visit your new site.

When we click on the link that says **your new site**, we should then be greeted with the most popular Drupal page (as shown in the following screenshot).



Welcome to your new Drupal website!

Please follow these steps to set up and start using your website:

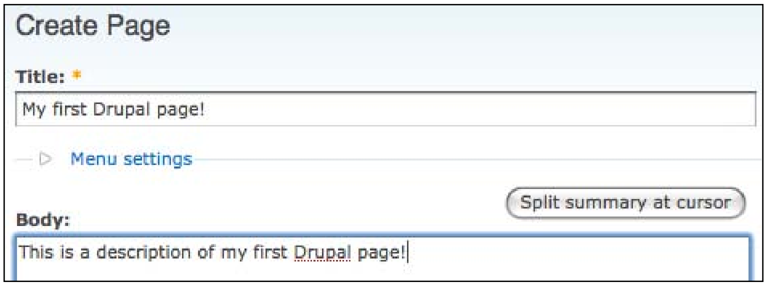
1. **Configure your website**
   Once logged in, visit the administration section, where you can customize and configure all aspects of your website.
2. **Enable additional functionality**
   Next, visit the module list and enable features which suit your specific needs. You can find additional modules in the Drupal modules download section.
3. **Customize your website design**
   To change the "look and feel" of your website, visit the themes section. You may choose from one of the included themes or download additional themes from the Drupal themes download section.
4. **Start posting content**
   Finally, you can create content for your website. This message will disappear once you have promoted a post to the front page.

For more information, please refer to the help section, or the online Drupal handbooks. You may also post at the Drupal forum, or view the wide range of other support options available.

With Drupal installed, we will now add some content.

Adding content to Drupal

In order to build a Flash application that utilizes Drupal content, we will first need to understand how to add content to Drupal. Fortunately, this step is very simple and first requires us to click on the link that says **Create content** on the left navigation menu. Once we click on this menu item, we should be given an option to either create a **Page** or a **Story**. A **page** is simply a single page of content. An example of a page would be if you were to create an "About" page for your web site that simply describes the nature of your business. A **Story** is a collection of pages that, together, combine and create a common piece of content. An example of a **Story** would be a web site tutorial. We can now create some content on our web site by clicking on the **Page** link. We can provide a **Title** and **Body** for our page, as follows:


Create Page

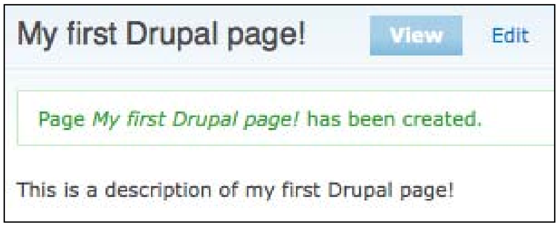
Title: *

My first Drupal page!

Body:

This is a description of my first Drupal page!

We can then click on the **Save** button, at the bottom of the page, to create our first page.


My first Drupal page!

Page *My first Drupal page!* has been created.

This is a description of my first Drupal page!

Now that we have created our first Drupal page, we are ready to move onto Flash.

Getting started with Flash

Adobe Flash is a multimedia platform that allows us to create animations and rich Internet applications. Each of these applications, created with Flash, is pre-compiled into the **Shockwave Flash** (**SWF**) file format. These applications can be embedded within a webpage or used on the desktop for local functionality. These SWF files are read and interpreted using the Flash Player, which must be installed as a plug-in for the browser viewing the page. This is rarely an issue though, since Flash is so widely used by so many web sites that it is rare to not have Flash installed on your visitor's browser.

So, our first task is to install Adobe Flash CS4 onto our computer, so that we are able to walk through the examples in this book. Although Flash CS4 is not a free application, you can easily download the 30-day trial version by going to `http://www.adobe.com/go/tryflash` and complete this book within that trial period. Once you have Flash CS4 installed on your machine, we are ready to move on.

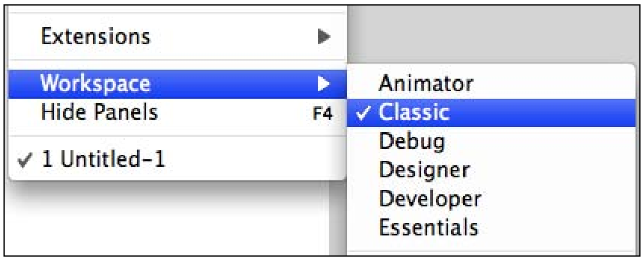
If you are using Flash CS3, you will still be able to use all of the tutorials in this book. Flash CS3 and CS4 are very similar, so there should not be any discrepancies between the two when following through the examples in this book.

Creating a new Flash project

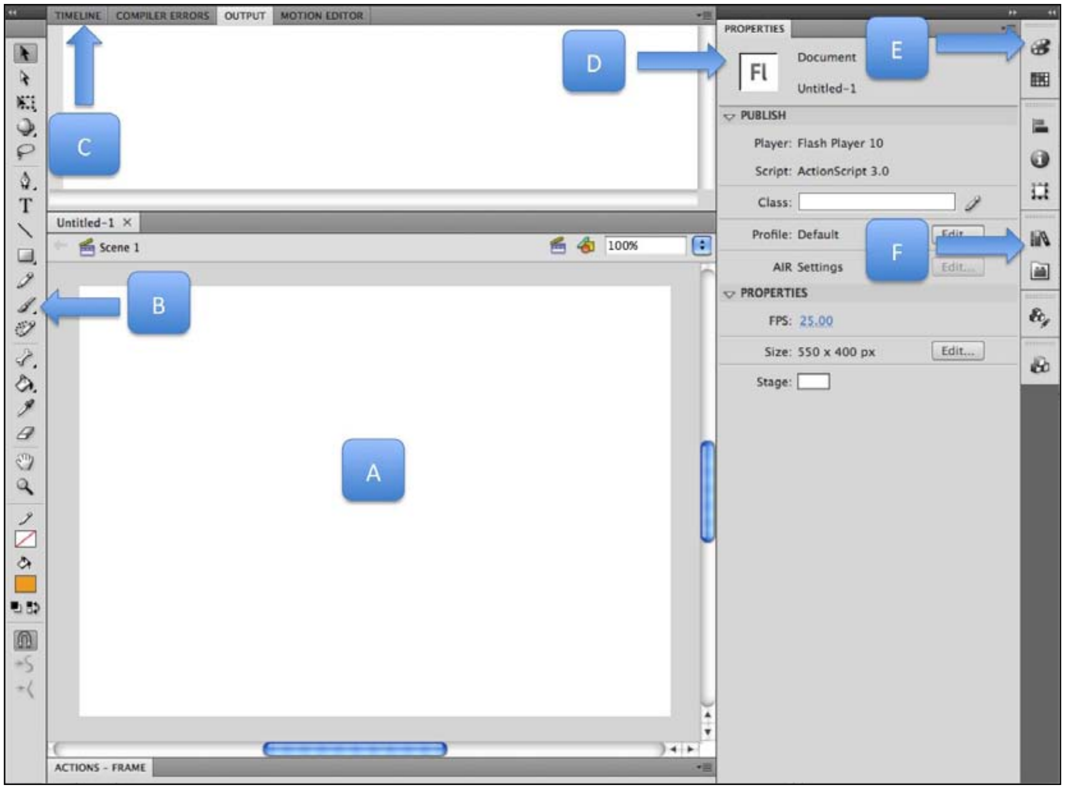
With Flash CS4 installed on our computer, we can open up the application and create a new project. When the application first comes up, you should see a welcome screen, where we can **Create a new project**. For all of the examples in this book, we will be using ActionScript 3.0 for our Flash applications. So, we will continue by clicking on the link that says **Flash File (ActionScript 3.0)**, which should then bring up an empty project. Since there are differences in the way that Flash CS3 and Flash CS4 look, our next task is to change the default workspace for Flash CS4 so that it looks like CS3. This will make it easier for everyone to follow every lesson in this book and keep any reference that I make regarding the location of certain tools consistent.

Setting up the workspace

In this step we will set the workspace layout so that it looks identical to the Flash CS3 default. In Flash CS4, we can accomplish this by clicking on the **Window** menu item. Once this menu item opens up, we will select the **Workspace** menu item, and then select **Classic**.



We should now have a workspace that resembles the following diagram. We will take a look at each highlighted section within this workspace, so that we know what they are when they are referenced in this book.



Player: Flash Player 10
Script: ActionScript 3.0
Class:
Profile: Default
AIR Settings
FPS: 25.00
Size: 550 x 400 px
Stage:

For each of the following sections, simply refer back to this diagram to see where to find the item under discussion.

A: The Stage

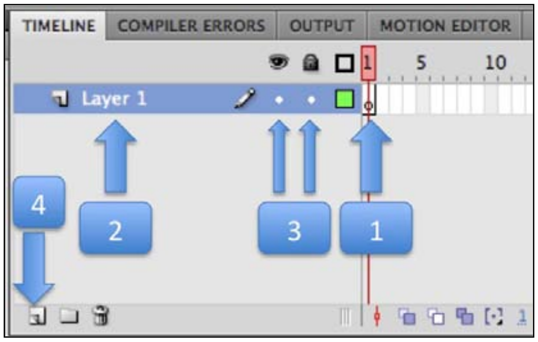
With our workspace set up the way that we want, we can shift our focus to the white square in the middle of the screen. This white square region is called the **stage** and will be referenced quite a bit in this book. The **stage** is simply the visible window for any object that is placed within our Flash application. By dragging an object off of the **stage**, we are removing the object from the visible region of our Flash application.

B: The Toolbar

The toolbar presents many of the tools that can be used to create or edit objects on the stage.

C: The Timeline

When you click on this tab, you should see the timeline for our Flash application. The timeline is a very important aspect in Flash since it allows us to create objects that can change in time. There are many elements within the timeline that will be important for us in this book, and each of these items is described as follows:



- Within the timeline, there are a series of white blocks that span the width of the timeline. Each of these little white blocks are called **frames**. A frame is a snapshot in time for our Flash application. We can combine a series of frames to create an animation (where an object moves for each new frame), or define different states for our objects, where each frame shows a different state for our object.
- Another important feature of the timeline is the ability to define multiple layers. We will use this quite often in this book since it not only allows us to control which objects are on top of other objects, but it also serves as a great way to organize our Flash applications for ease of use.
- Having each object within its own layer also gives us the power to hide or lock each layer. This can be done by clicking on the dot within the 👁 column (visibility) or the 🔒 column (lock). This will help us to keep other objects unaffected when we are working on a different layer.
- We can also create new layers by clicking on the ▣ symbol. Once we have created a new layer, it is best practice to name that layer by clicking on the name until it turns into an edit box, where we can then provide any name we would like.

D: The Properties panel

The properties panel is used to describe the object that is currently selected. We will use this panel quite often in order to change the width and height, and to give names to our object's instance.

E: The Color Palette

We will use this tool to change the fill colors for the objects that we create. We will also use this tool to create gradients to give our application depth and character.

F: The Library

This tool is used to keep track of all objects that we have created within our Flash application. We will use this to edit existing objects as well as to create new objects that will be used in our Flash application.

Now that we are familiar with the Flash **Integrated Development Environment** (**IDE**), we can create something within Flash and then embed that within Drupal.

Creating a Flash application

For this example, our Flash application will be very basic. For the most part I would just like us to walk through the process of creating a Flash application within Flash, and then take that application and embed it within a page in Drupal. Once we have conquered this, we will be geared up to create some really cool applications that will surely wow your visitors. But keep in mind that your imagination is the key, so feel free to go crazy and create something very cool, and not feel limited with what we create in this section.

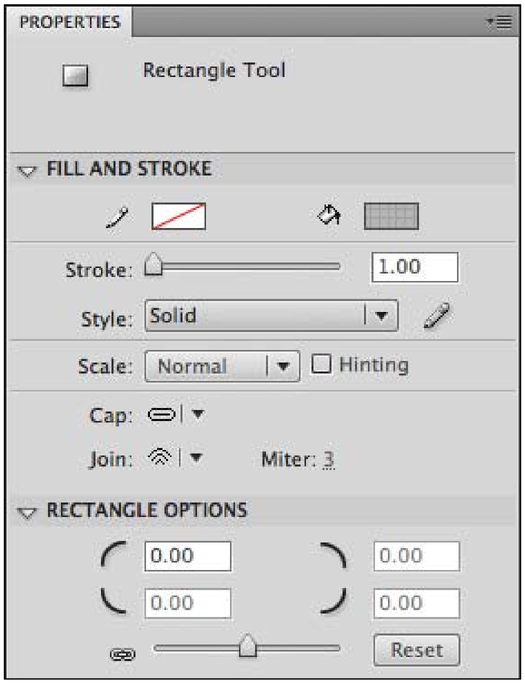
Before we begin, however, we will need to create a home for our Flash project. In this book we will always start out each chapter by first creating a new directory to hold the contents for that chapter. With that said, we first need to create a new folder called `chapter1` and then save our currently opened up Flash application as `chapter1.fla` within this folder, by clicking on the **File | Save As** menu item. Once we have saved our Flash application, we can now start by adding objects to our stage.

Creating a background

Our first task will be to create a background for our Flash application so that it sticks out when viewing through Drupal. We can do this by clicking on the □ tool within our toolbar. This should then change our mouse to the cross-hair symbol when you move the mouse over the stage. Before we begin to draw our rectangle, though, we will need to make sure that all of our rectangle properties are set up just the way we want them.

Rectangle properties

We will now shift our focus to the **PROPERTIES** panel, which should show some of the options that we have for drawing a rectangle object.



Stroke: 1.00

Style: Solid

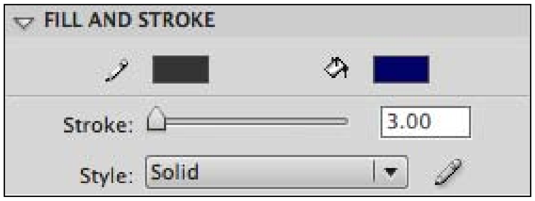
Scale: Normal ☐ Hinting

Miter: 3

0.00 0.00

0.00 0.00

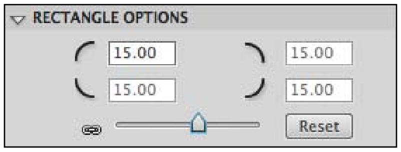
The **FILL AND STROKE** region is used to describe how the rectangle will be drawn and what colors can be used to either fill or stroke (border) the rectangle. For this example we would like to have a rectangle that has a 3 pixel dark grey border, with a dark blue center, which we can configure by setting the following:



Stroke: 3.00

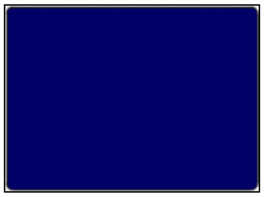
Style: Solid

Finally, we would like our rectangle to have rounded corners. We can do this by adding a value within the **RECTANGLE OPTIONS** section that shows the symbols. To have a 15 pixel radius for the rounded corners, we can provide the following:


RECTANGLE OPTIONS

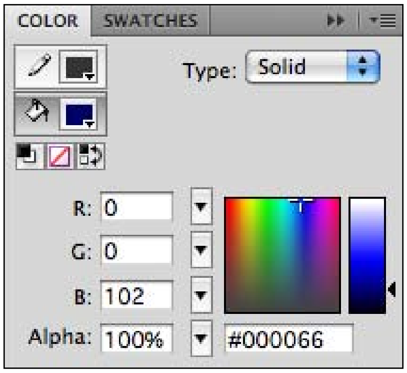
15.00 15.00
15.00 15.00

We can now move our cursor over the upper left-hand corner of our stage region (where the white square begins), and draw our rectangle region as follows:



Adding a gradient

We will now give our background some character by adding a gradient as the fill color. To do this, we will first click on the fill region of the rectangle using the Selection tool (), and then open up the Color tool on the right side of the screen.

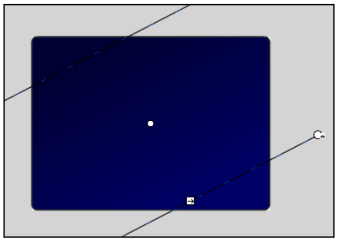


Type: Solid

R: 0
G: 0
B: 102
Alpha: 100%
#000066

We can now give our background a linear gradient by clicking on the **Type** drop-down menu and selecting **Linear**. Next, change the gradient colors by first clicking on the right tab within the Gradient tool , and then changing that color back to the dark blue color. Then, click on the left tab of the gradient tool and give it a lighter blue color.

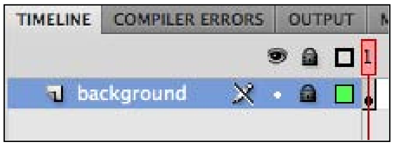
Finally, to make this background more interesting, we will orient the gradient so that it is not completely horizontal. We can do this by first clicking and then holding down our mouse over the Transform tool (⬚). By clicking and holding down our mouse over this tool, it should bring up a submenu, where we can then select the Gradient Transform tool (▣). Once we have this tool selected, we will click on our fill region, which will expose some handles where we can resize and change the orientation of our gradient. The circle symbol (↻), over our fill region, will allow us to change the orientation of our gradient, which we will use to change our fill gradient to be oriented at approximately 45 degrees.



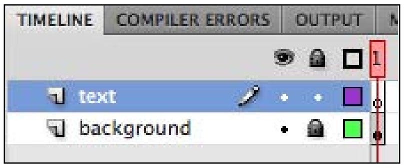
We are now done with our background, and ready to move onto adding some text.

Adding text to a Flash application

Before we begin adding text, we first need to create a new layer so that our text does not collide with our newly created background. To do this, we will first click on our **TIMELINE** tab, where we will first rename the default layer name to **background**, and then lock this layer by clicking on the dot in the 🔒 column.



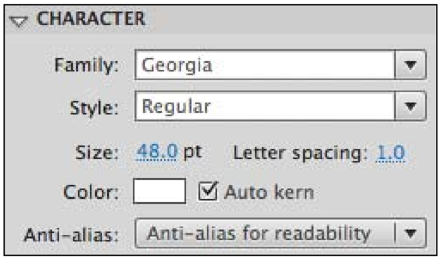
After we have done this, we can create a new layer on top of the background layer by clicking on the ☐ within the bottom layer tool menu ☐ ☐ 🗑, and then calling this new layer as **text**.



Now that we have a blank layer to add our text, we can click on the Text tool (T) within our toolbar and, and shift our focus to the **PROPERTIES** panel.

Text properties

The text properties are most likely familiar to anyone who has used a Word processor application. It allows you to change the color, size, style, and letter spacing. In our example, we want a fairly large size and light color so that our text will stand out against the dark background that we just created. For that reason, we should provide the following information for our text field properties:



CHARACTER

Family: Georgia

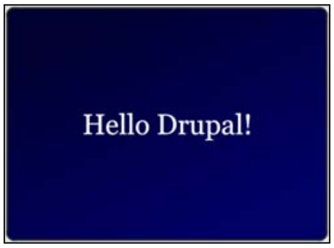
Style: Regular

Size: 48.0 pt Letter spacing: 1.0

Color: □ ☑ Auto kern

Anti-alias: Anti-alias for readability

After we have our properties set up, we can add our text to our stage by clicking anywhere on the stage we would like to show our text. When we are done, we should see the following:



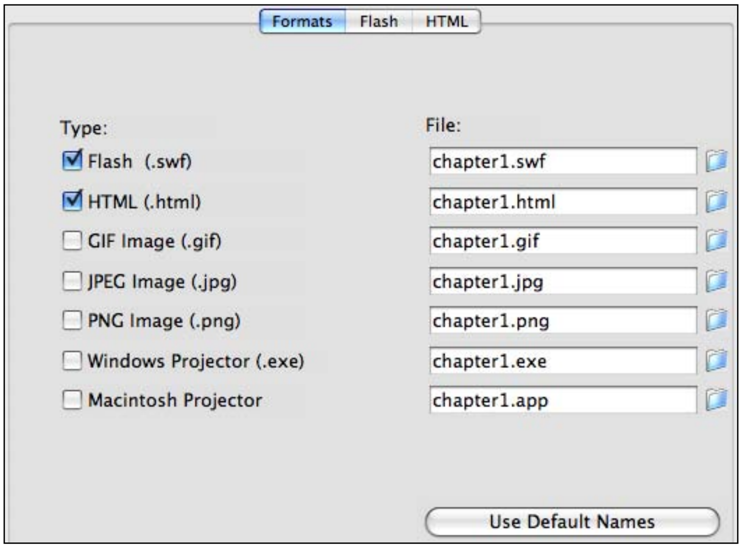
Hello Drupal!

Compiling our Flash application (making a SWF)

Now that we have created our Flash application, the next step is to run and compile this into a SWF file, so that we can embed it within our Drupal web site. This will first require us to take a look at the **Publish Settings**, so that we can make the necessary changes to our published SWF file.

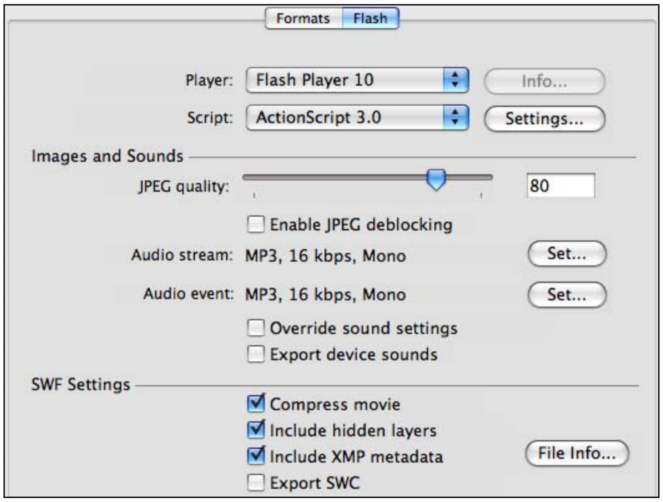
Publish Settings

Publish Settings can be found by clicking on the **File | Publish Settings** menu item, where we will see the following page:



Type:	File:
☑ Flash (.swf)	chapter1.swf
☑ HTML (.html)	chapter1.html
□ GIF Image (.gif)	chapter1.gif
□ JPEG Image (.jpg)	chapter1.jpg
□ PNG Image (.png)	chapter1.png
□ Windows Projector (.exe)	chapter1.exe
□ Macintosh Projector	chapter1.app

Since we are using Drupal to show our Flash applications, the first thing that we need to do is uncheck **HTML** from the publish type column. After we have done this, we can take a look at the **Flash** tab on the publish settings, which will show the following screen:



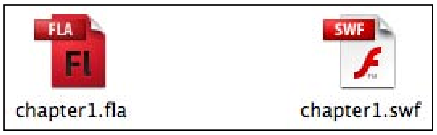
Player: Flash Player 10
Script: ActionScript 3.0

Images and Sounds
JPEG quality: 80
□ Enable JPEG deblocking
Audio stream: MP3, 16 kbps, Mono
Audio event: MP3, 16 kbps, Mono
□ Override sound settings
□ Export device sounds

SWF Settings
☑ Compress movie
☑ Include hidden layers
☑ Include XMP metadata
□ Export SWC

Since our goal is to stay consistent with the Flash CS3 readers, we will probably want to use Flash Player 9 to compile and run our SWF movies. We can do this by clicking on the **Player** drop-down box and then selecting **Flash Player 9**. When we are done, we can click on the **OK** button at the bottom of the settings page to accept the changes.

Now that we have our settings in place, we can run and compile our Flash application by pressing ⌘+*Enter* for Mac, or *Ctrl*+*Enter* for Windows. If you wish to just compile your application, then you can alternatively select **File | Publish** from the main menu. When we are done with this step, we should be able to navigate to our `chapter1` folder and see the following:



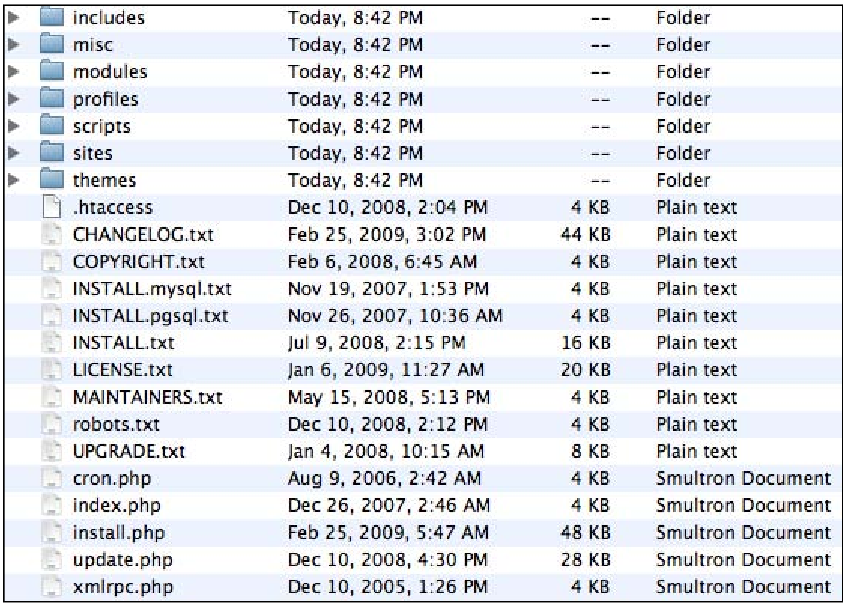
The SWF file is the compiled Flash file that we will now embed within our Drupal web site.

Adding Flash content to Drupal

There are many ways to add Flash content to Drupal, but what I am going to cover in this section is what I have found to be the easiest. Thanks to the wonderful Drupal community, there is already a fantastic module that was built to easily add Flash applications to Drupal. This module is called FlashNode and can be found at `http://www.drupal.org/project/flashnode`. Within this project page, we need to download the latest version for Drupal 6. We can determine a module's Drupal version by taking a look at the first number in the package version, where 6.x means it is for Drupal 6. After we have downloaded the version build for Drupal 6, which was version 6.x-3.1 at the time of writing, we can install this module in our local host Drupal installation.

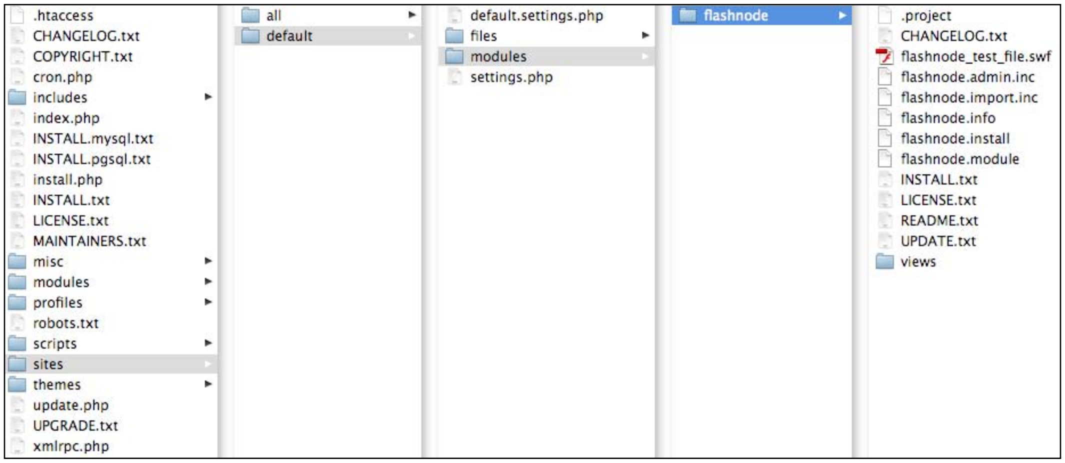
Installing a contributed Drupal module

The first thing that we need to do to install a Drupal module is locate our Drupal installation that we placed in our computer earlier in this chapter. Within this directory, we should see the following files:

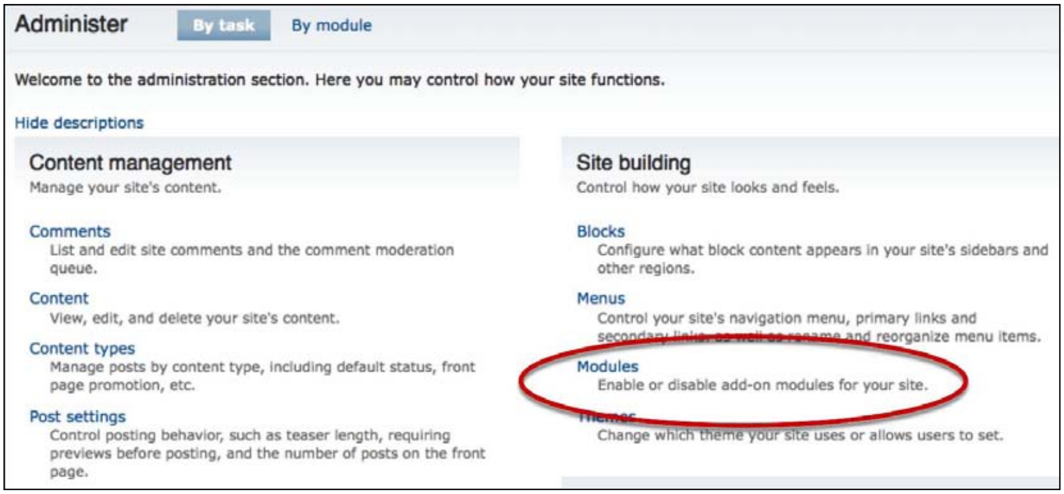


includes	Today, 8:42 PM	--	Folder
misc	Today, 8:42 PM	--	Folder
modules	Today, 8:42 PM	--	Folder
profiles	Today, 8:42 PM	--	Folder
scripts	Today, 8:42 PM	--	Folder
sites	Today, 8:42 PM	--	Folder
themes	Today, 8:42 PM	--	Folder
.htaccess	Dec 10, 2008, 2:04 PM	4 KB	Plain text
CHANGELOG.txt	Feb 25, 2009, 3:02 PM	44 KB	Plain text
COPYRIGHT.txt	Feb 6, 2008, 6:45 AM	4 KB	Plain text
INSTALL.mysql.txt	Nov 19, 2007, 1:53 PM	4 KB	Plain text
INSTALL.pgsql.txt	Nov 26, 2007, 10:36 AM	4 KB	Plain text
INSTALL.txt	Jul 9, 2008, 2:15 PM	16 KB	Plain text
LICENSE.txt	Jan 6, 2009, 11:27 AM	20 KB	Plain text
MAINTAINERS.txt	May 15, 2008, 5:13 PM	4 KB	Plain text
robots.txt	Dec 10, 2008, 2:12 PM	4 KB	Plain text
UPGRADE.txt	Jan 4, 2008, 10:15 AM	8 KB	Plain text
cron.php	Aug 9, 2006, 2:42 AM	4 KB	Smultron Document
index.php	Dec 26, 2007, 2:46 AM	4 KB	Smultron Document
install.php	Feb 25, 2009, 5:47 AM	48 KB	Smultron Document
update.php	Dec 10, 2008, 4:30 PM	28 KB	Smultron Document
xmlrpc.php	Dec 10, 2005, 1:26 PM	4 KB	Smultron Document

If we open up the `sites` folder, we should then see an `all` and a `default` folder. These folders are used to separate the files, modules, and themes for any web site that is sharing the single Drupal installation (called multisite). For example, if we wish to install a module for all of our web sites, then we would place this new module within the `all` folder. However, if we just wish to include this module in the default site (which is the site we are using right now), then we will have to place the module within the `default` folder. In a typical multisite configuration, there would be a separate folder within the `sites` directory for each web site that is being run off the single Drupal installation. But for our purposes, we will use the `default` folder for any additional module that we install to our Drupal web site. So, within the `default` folder, we will need to create a new folder called `modules`, and then place the extracted contents of the **flashnode** module within this directory as follows:



Now that we have the **flashnode** module in the right location on our server, we can navigate to our Drupal web site by going to `http://localhost/drupal6`. Once we are there, we can go to our administrator section by clicking on the **Administer** link on the left menu. Once we are within our Drupal administration section, we will click on the **Modules** link. Throughout this book I will refer to this type of navigation as saying "navigate to **Administer | Modules**".



Welcome to the administration section. Here you may control how your site functions.

Hide descriptions

Content management

Manage your site's content.

Comments
List and edit site comments and the comment moderation queue.

Content
View, edit, and delete your site's content.

Content types
Manage posts by content type, including default status, front page promotion, etc.

Post settings
Control posting behavior, such as teaser length, requiring previews before posting, and the number of posts on the front page.

Site building

Control how your site looks and feels.

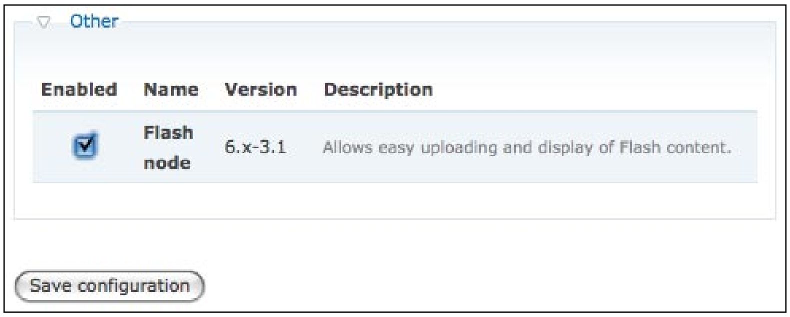
Blocks
Configure what block content appears in your site's sidebars and other regions.

Menus
Control your site's navigation menu, primary links and secondary links, as well as rename and reorganize menu items.

Modules
Enable or disable add-on modules for your site.

Themes
Change which theme your site uses or allows users to set.

Once we are within the **Modules** section, we can scroll down to the bottom of the page and enable the **FlashNode** module by clicking on the checkbox next to this module, and then clicking on the **Save Configuration** button at the bottom of the page.

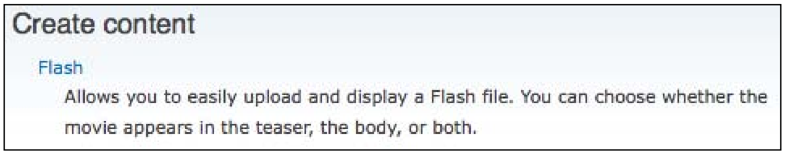


▽ Other

Enabled	Name	Version	Description
☑	Flash node	6.x-3.1	Allows easy uploading and display of Flash content.

Adding Flash!

We are now ready to add our Flash application to Drupal. We will first click on the **Create Content** link from the left navigation menu, and then select **Flash** from the list of items.

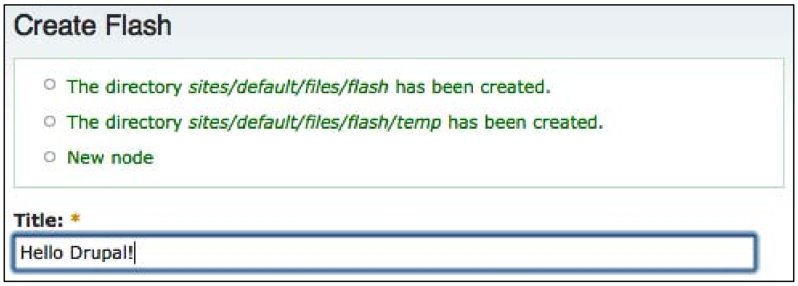


Create content

Flash

Allows you to easily upload and display a Flash file. You can choose whether the movie appears in the teaser, the body, or both.

We will then see the **Create Flash** page, where we will start by giving our new Flash node a title.



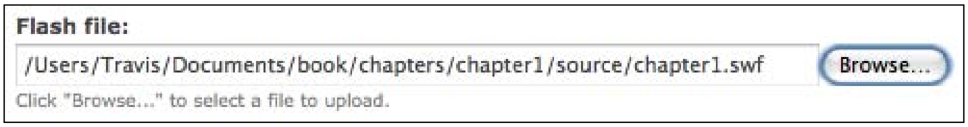
Create Flash

- The directory *sites/default/files/flash* has been created.
- The directory *sites/default/files/flash/temp* has been created.
- New node

Title: *

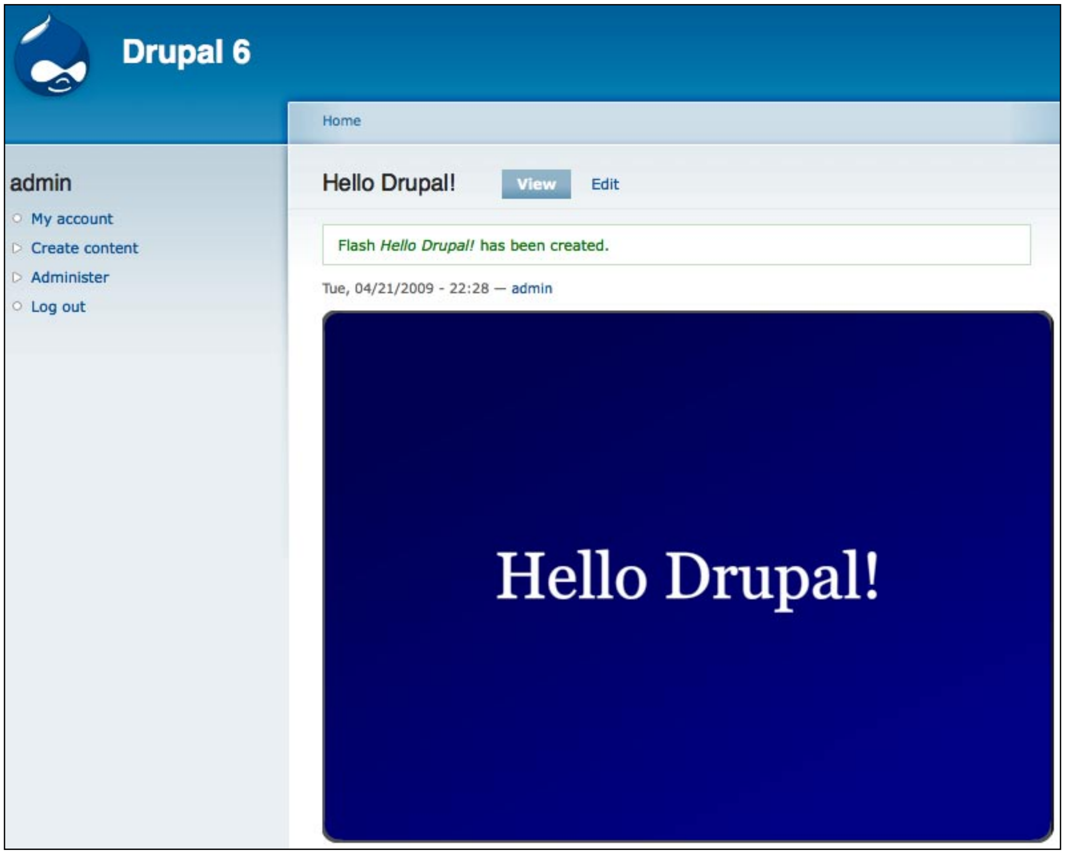
Hello Drupal!

After this, we will select the Flash file that we just created by clicking on the **Browse** button next to the **Flash file** input field, and then selecting our SWF file using the file browser window.



Flash file:
/Users/Travis/Documents/book/chapters/chapter1/source/chapter1.swf
Click "Browse..." to select a file to upload.

Finally, we can scroll down to the bottom of the page, where we can click on the **Save** button. Congratulations, you have added Flash to Drupal!



Flash *Hello Drupal!* has been created.

Tue, 04/21/2009 - 22:28 — admin

Summary

We now have the foundation for building some dynamic applications for Drupal. We can easily use the lessons learned in this chapter to build some very cool Flash widgets to be used within Drupal. But these applications would be limited, since they will be confined to only showing the content provided from within that Flash application. In the following chapters we will dive into how to incorporate Drupal content within our Flash applications. This will open up a whole new world of functionality that we can utilize to create some very exciting Flash applications.

2

Building a "Hello World" Application

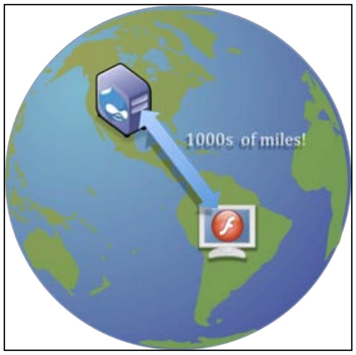
We have built a static Flash application and integrated it within Drupal, now our next step is to replace the static text with dynamic content extracted from Drupal. We will accomplish this task by building a Flash application that is able to read a Drupal "Hello World" page, and then displaying that text using Flash. Although this may seem like a trivial task, surprisingly, there is much effort required to simply display those two words using Flash and Drupal. In this chapter we will go over all the necessary elements needed for communication between Flash and Drupal. Some of the key topics that will be covered are as follows:

- How Flash and Drupal communicate
- Understanding Web Services
- Setting up Drupal for Web Services using the Services module
- Using Drupal's web services to say "Hello World" in Flash
- Using FlashVars in our Flash applications

How Flash and Drupal communicate

In order to understand how Flash and Drupal communicate, it is important to note how each one differs in where it is executed. When we build a Flash application for Drupal, the loading and execution of our Flash application will be completely different from the execution of our Drupal website. As with any Drupal website, the execution of all the code within Drupal will be performed on the server in which it resides. Once this code is executed on the server, data is sent in the form of HTML to the machine that the visitor is using to view our website. Any Flash application that we build on top of Drupal is sent along with this HTML, and then loaded and executed on the visitor's remote machine. In web terminology, this association is called Client and Server, where Drupal will be the server-side application and the Flash application will be the client-side application.

The catch here is that there is no guarantee that the server and client machines are near each other physically. They could be a couple of feet apart or thousands of miles away from each other, as seen in the following illustration:



Because of this, communication between Drupal and Flash requires the use of a standard protocol, commonly referred to as web services.

Understanding web services

The fundamentals of web services are structured around the concept of remote function calling—or remoting. Using the power of XML, remoting allows one application to remotely call a function within another remote application. What this means is that one application that is running on a computer half way around the globe can send a request to call a function on your home computer. This process is commonly referred to as **RPC** (**Remote Procedure Call**). (To read more about RPC, visit `http://en.wikipedia.org/wiki/Remote_procedure_call`.) There are many different implementations of RPC, but all of them accomplish the same task, that is, of breaking up the elements that define a function into a standardized XML format. This XML is then transmitted to the remote location where it can be interpreted and executed as if the two procedures were located on the same box or even within the same application. The detailed inner workings of RPC are slightly out of the scope of this book. Anyway, what is important is for you to understand is how two separate applications can accomplish this technique of RPC—since this is exactly how Flash and Drupal communicate.

Let me demonstrate this by giving you a very simple Flash application that prints (you guessed it!) "Hello World". Let's go ahead and open up our Flash application and start a new project. In this project, we will start by simply adding a `trace` statement to print "Hello World" in our debug shell. The `trace` statement is used within Flash to print out any string or variable in the Output window. This is a very handy tool because it lets us visually see when pieces of code are executed, and the values of certain variables when that execution was made. Within the **Actions** panel, we can now see how the trace statement works by placing the following command:

```
trace("Hello World!");
```

Once this code is in place, we can run our Flash application by pressing ⌘+*Return* for Mac or by pressing *Ctrl* + *Enter* for Windows. This should then bring up a blank screen where we will see the text "Hello World!" within the Output panel.

Now, let's take that simple line of code and turn it into a function that does the same thing. It should look like the following:

```
function sayHello()
{
   trace( "Hello World" );
}
```

If we run the Flash application again, we should see that the code we wrote doesn't do anything. The reason for this is that the simple declaration of a function really doesn't do anything unless someone calls that function. We can do this by placing the call just below the function declaration like the following:

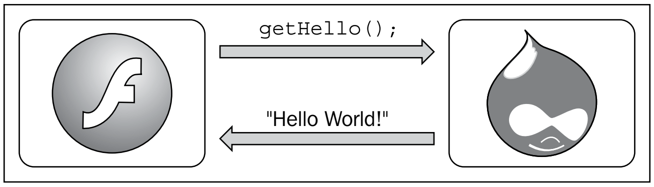
```
function sayHello()
{
   trace( "Hello World" );
}
sayHello();
```

So, in the last example, we have the same application defining the routine and making the call. In web services, these two will be separated between two completely different applications. One application will define the function `sayHello`, and the other application will be making the remote procedure call.

The previous example does a fair job of showing you the concept of how web services work. But, it isn't very accurate in showing you how we plan to work with Flash and Drupal. In this scenario, we will use Drupal as the source of the content and Flash as the mechanism for displaying that content. Because of this, we need to change our above code so that the text "Hello World" is retrieved instead of assumed. Still working within a single application, this would look like the following:

```
function sayHello( hello:String )
{
   trace( hello );
}
function getHello() : String
{
   return "Hello World!";
}
sayHello( getHello() );
```

I know that this code seems very bloated for simply printing "Hello World", but we are getting much closer to a real-life web service interaction between two remote applications. In this example, Flash will call `getHello`, which will end up making a remote function call into Drupal and then populate the `sayHello` argument with whatever it returns, as shown:



But, there is still something wrong with this example. Upon closer observation, it is clear that we are assuming that the `getHello` routine will instantly return the text from its content source. In a web service scenario, this `getHello` routine will be calling a remote location that will always take an unknown amount of time to return something, if anything at all. This type of interaction is called "asynchronous" and is usually the cause of much confusion for a programmer who is just learning this process.

Synchronous versus asynchronous programming

In the previous example, you will notice that there are two functions. The `sayHello` function is used to print whatever argument it receives, while the other function is used to retrieve and return the text "Hello World". Now, combine these two functions and you will get an overly complicated way to print "Hello World". However, this over complication is necessary to understand how two independent pieces of software interact. Of course, for the sake of simplicity, I have chosen to keep these two functions in the same application, but soon we will separate the two between Flash and Drupal. So, in this example, I would like to concentrate our attention on the `getHello` function.

```
function getHello() : String
{
   return "Hello World";
}
```

This function, as it is designed right now, is very simple because it is programmed synchronously. This means that the time taken by this function to retrieve the string "Hello World" is deterministic. I am able to guarantee that after we execute this function, we will end up with a string that says "Hello World". But this is not how it works in the world of web services. In a web service, or remote procedure call, this function will send a request to an external application to get this string. Also, what is even more important to realize is that we don't know if and when this external application will return our string. Because of this, we need to design this function asynchronously. Let me give you an analogy…

> To best describe how an asynchronous software system works, I would like to give you a real-life analogy: My wife asks me to take out the garbage every now and then. If we talk in software terms, we could say that my wife is an application, telling another application (me), to perform its routine, that is, `takeOutGarbage`. Now, when my wife sends out this command, there is really no telling if and when she will get the results of me actually taking out the garbage. It could be hours, days, or maybe never. But this does not mean that my wife just waits until I am finished with my task—since she could be waiting for a really long time and lose productivity in getting other things done. Instead, she goes about doing other things, and when I finally do take out the garbage, she then gets a notification from me that I have finished my work. This is exactly how asynchronous software behaves.

So, let's change our `getHello` function into an asynchronous routine. In order to accomplish this task, we need to utilize a `callback` function. A `callback` function is just a way to tell any piece of code "Hey, do this, and then let me know when you are finished". The `callback` function allows you to receive notification when something has completed, which then allows any software process to perform other tasks while it waits for notification. Here is an example of how to pass a `callback` function into our `getHello` function:

```
function sayHello( hello:String )
{
   trace( hello );
}
function getHello( callback:Function )
{
   callback( "Hello World" );
}

getHello( sayHello );
```

When we execute this Flash application, we will see that the results are surprisingly the same. The only difference is, this time, we are using a `callback` function that works by passing the function `sayHello` into the `getHello` function as an argument or (`callback`). When the `getHello` routine is finished doing what it needs to do, it will call the `callback` function with the correct text. For us, this is just the same thing as calling the `sayHello` function with the string "Hello World".

The good news here is that the getHello function can now take all the time it needs. In fact, we can now completely change the code within getHello to call an external web service routine and not have to worry about our application hanging while it waits for a response. This is the main benefit of using callback functions, as they provide one of the main ingredients for asynchronous programming. So, let's put this into action by setting up Drupal for web services.

Setting up Drupal for web services

In order to handle web services within Drupal, several contributors took an initiative to develop a series of fantastic modules that'll accomplish the task of remote communication. These modules are all subsets of the module Services, which can be found at http://www.drupal.org/project/services. Utilizing the power of this module, external applications are given the ability to make remote calls to Drupal and extract the data that they need. So, our first task in making a Flash application that says "Hello World" within Drupal will be to set up our Drupal installation so that it is ready for external interaction. And we will do this using the Drupal Services module.

Installing and configuring the Services module

The first task in getting web services connected for Drupal is to install the Services module. You can find this module located at http://www.drupal.org/project/services. Once you are there, you will be given the option to download a version for either Drupal 5 or Drupal 6. This book assumes you are using Drupal 6, but since the Services module works the same way on Drupal 5 as it does on 6, you should be able to follow along, regardless of which version of Drupal you are using. Once you have downloaded Services, you will need to extract the contents of this TAR file in your sites/default/modules folder, which we created in the previous chapter. After these files are in place, you will notice that there are two subfolders within the main Services directory called Servers and Services.

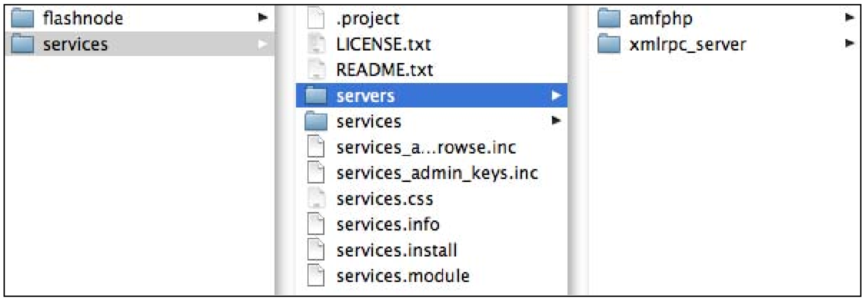
Servers and Services

Services are the software elements that define the routines that can be executed by outside applications. By enabling any of these modules, you are now exposing a series of routines within their corresponding modules as "executable", so that any outside application can use them to retrieve or set data within Drupal.

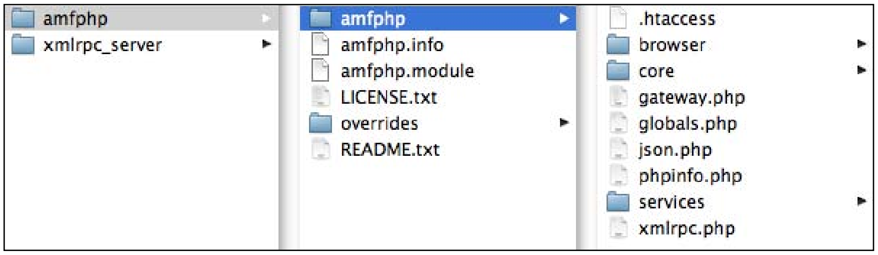
The Servers act as the translators that take the XML format of the service request from an outside source, and then translate that XML into the execution of an internal routine. They also handle the responses (or returns) from those routines, and convert them back into the XML format to be sent back to the external application that originally made the request. There are several different types of servers that can be used for RPC communication, but the one that we will need to use for our ActionScript 3 interaction is called **AMFPHP** (**Action Message Format PHP**).

Installing AMFPHP

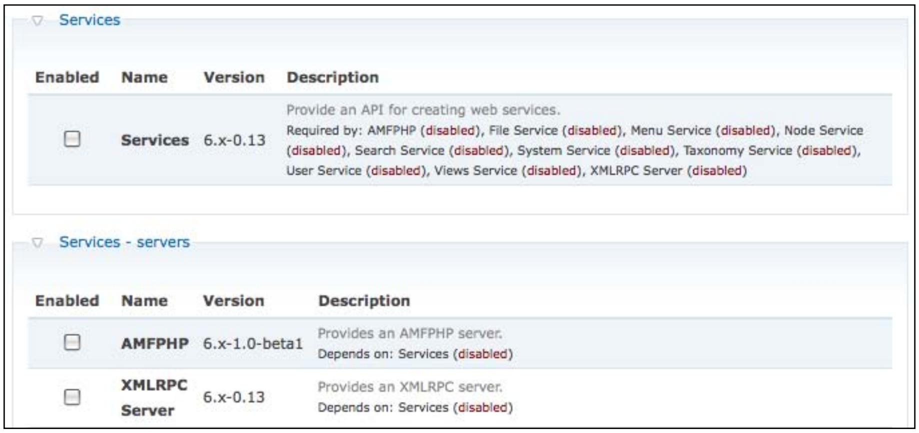
AMFPHP is a remoting gateway to be used specifically for Flash and ActionScript, and Drupal already has a module that creates an AMFPHP server plug-in for the Services module. This can be found at `http://www.drupal.org/project/amfphp`. Since this is a server that will be used with the Services module, we can place the contents of this download within the **servers** folder in the **services** module directory.



But before we install this module, we will need to make sure that we download the AMFPHP source files located at `http://www.amfphp.org`. Once we have downloaded the AMFPHP server package, we will need to place the contents of that package within an **amfphp** folder in our AMFPHP module folder as shown in the following illustration:



After we have both the AMFPHP and Services modules in place, we can then navigate to the **Administer | Modules** section of your website and take a look at the **services** section, which includes the **Services** module, **servers**, and **services**.

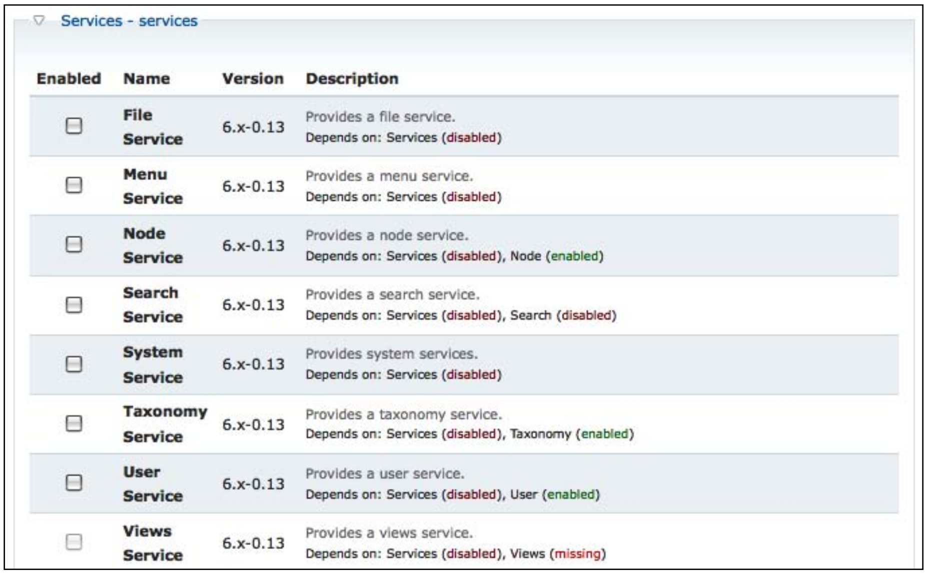


▽ Services

Enabled	Name	Version	Description
☐	**Services**	6.x-0.13	Provide an API for creating web services. Required by: AMFPHP (disabled), File Service (disabled), Menu Service (disabled), Node Service (disabled), Search Service (disabled), System Service (disabled), Taxonomy Service (disabled), User Service (disabled), Views Service (disabled), XMLRPC Server (disabled)

▽ Services - servers

Enabled	Name	Version	Description
☐	**AMFPHP**	6.x-1.0-beta1	Provides an AMFPHP server. Depends on: Services (disabled)
☐	**XMLRPC Server**	6.x-0.13	Provides an XMLRPC server. Depends on: Services (disabled)



▽ Services - services

Enabled	Name	Version	Description
☐	**File Service**	6.x-0.13	Provides a file service. Depends on: Services (disabled)
☐	**Menu Service**	6.x-0.13	Provides a menu service. Depends on: Services (disabled)
☐	**Node Service**	6.x-0.13	Provides a node service. Depends on: Services (disabled), Node (enabled)
☐	**Search Service**	6.x-0.13	Provides a search service. Depends on: Services (disabled), Search (disabled)
☐	**System Service**	6.x-0.13	Provides system services. Depends on: Services (disabled)
☐	**Taxonomy Service**	6.x-0.13	Provides a taxonomy service. Depends on: Services (disabled), Taxonomy (enabled)
☐	**User Service**	6.x-0.13	Provides a user service. Depends on: Services (disabled), User (enabled)
☐	**Views Service**	6.x-0.13	Provides a views service. Depends on: Services (disabled), Views (missing)

Although we will explore and utilize most of these services in this book, in this chapter we will only be concerned with enabling the System and Node Services. These services will allow us to log in to Drupal and extract node information for any given node ID in the Drupal system. With that said, let's go ahead and enable the **Services**, **AMFPHP**, **System Service**, and **Node Service** modules. Once we have these modules enabled, we can go to **Administer | Services** to learn how to configure the **Services** module for external communication.

Services configuration

Once we navigate to **Administer | Services**, we should see three different tabs at the top of the screen: **Browse**, **Keys**, and **Settings**.

Under the **Browse** tab, we should see a listing of all Servers installed as well as all the Services available to outside applications. Since we have enabled the System and Node Services, we should see just a system and node section followed by the following routines associated with those services:

- `system.connect`
- `system.mail`
- `system.getVariable`
- `system.setVariable`
- `system.moduleExists`
- `node.get` (which is `node.load` in Drupal 5)
- `node.save`
- `node.delete`

By using these functions, outside applications can connect to your Drupal system and read, write, or manipulate any node in your Drupal website. Although this may raise a red flag of security, keep in mind that much effort has been contributed to the security of the Services module, which is what brings us to the next couple of tabs: **Keys** and **Settings**.

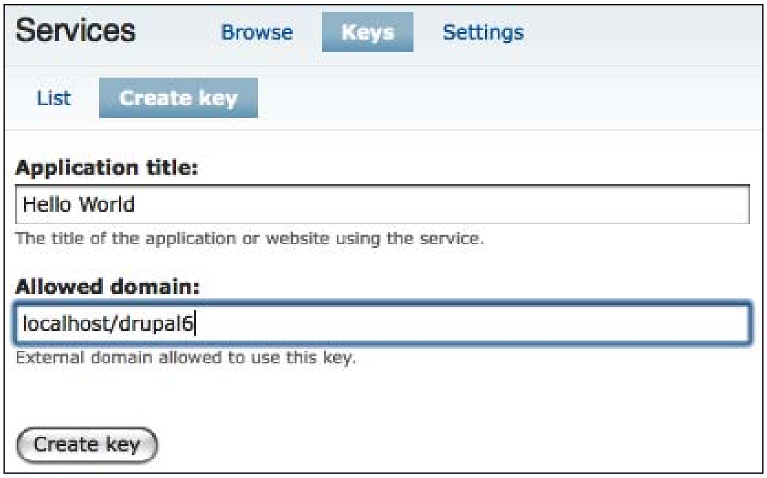
Creating a Services key

If we click on the **Keys** tab, we should be given two tabs at the top, where we can either list the valid keys in our system, or create a new one. Only the service routines that manipulate data in our Drupal system require the application using that service to provide an **API** (**Application Programming Interface**) key before any data manipulation can occur. The concept of a key is rather simple. When we create a new key, the Services module will create a randomly generated string. Then, we can provide this string within our external application to call certain routines that manipulate data. This is just a way of protecting our site from malicious spam bots bombarding our server with service calls and attempting to manipulate our node data automatically. So, let's go ahead and create a new key by clicking on the **Create Key** tab.

Once we click on this tab, we will be given the option to enter text in two different fields: the **Application title**, and the **Allowed domain**.

The **Application title** can be anything, so we will enter something descriptive about the application we are building, which will be **Hello World** for our example. The next field is an important one.

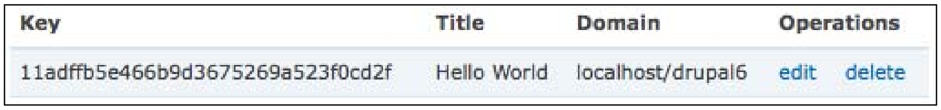
The **Allowed domain** will be used to populate a cross-domain file on your server that specifies the domains allowed to access the services on our server. Since we are using a local server to test our application, we can provide the domain name of our localhost plus any subdirectory where our Drupal root is located. For our example, this will simply be **localhost/drupal6**.



Application title:
Hello World
The title of the application or website using the service.

Allowed domain:
localhost/drupal6
External domain allowed to use this key.

When we are done with entering these values, we can click on the **Create Key** button to create our key. After our key has been created, we should see our new key added to the list of valid keys to be used by external applications as shown in the following screenshot:

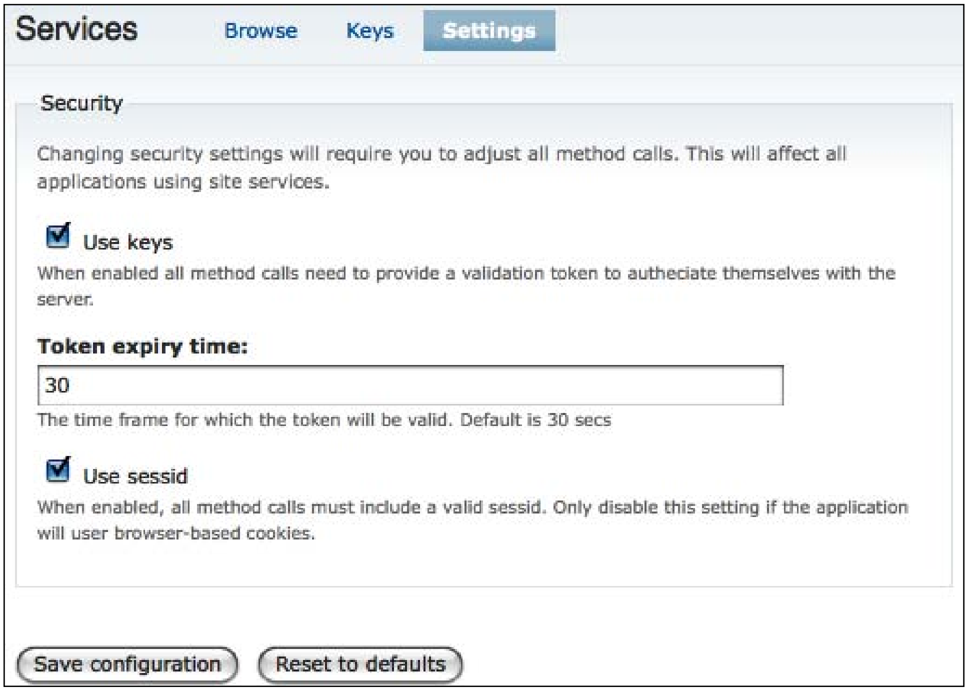


Key	Title	Domain	Operations	
11adffb5e466b9d3675269a523f0cd2f	Hello World	localhost/drupal6	edit	delete

Now that we have an API key, we are ready to move on to the **Settings** tab.

Services settings

The **Settings** tab of the Services administrator is used to enable or disable certain security measures as well as increase the timeout used when calling routines that require an API key. The only thing that is important to note here is that you should always have both the **Use keys** and the **Use sessid** (Session Handling) checked at all times when dealing with Services. Otherwise, you run the risk of forgetting to re-enable them later, which will introduce a security risk to your site. It also forces you to write your applications properly so that they will never compromise with the security of your data.



Security

Changing security settings will require you to adjust all method calls. This will affect all applications using site services.

☑ Use keys

When enabled all method calls need to provide a validation token to autheciate themselves with the server.

Token expiry time:

30

The time frame for which the token will be valid. Default is 30 secs

☑ Use sessid

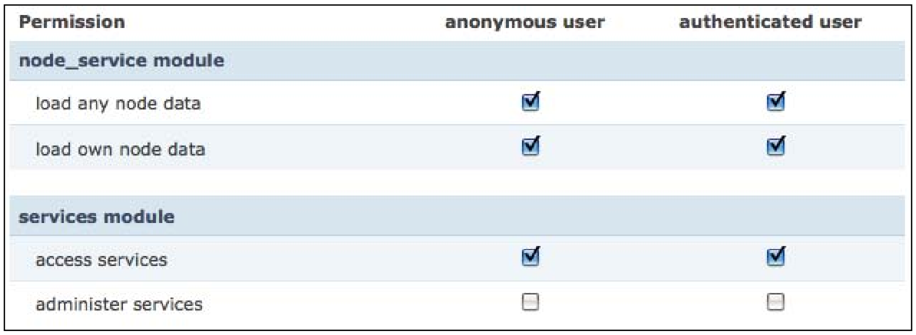
When enabled, all method calls must include a valid sessid. Only disable this setting if the application will user browser-based cookies.

So, now that we have the Services configured, our next step is to explore the user permissions used to control access of the web services to different user roles in your Drupal system.

Service Permissions

Before we can start using our web service routines to build our Hello World application in Flash, it is also very important to configure the permissions so that we can access the data that we need from Drupal. We do this by first going to the **Administer | Permissions** section within the **User Settings** section of the Drupal Administrator.

Once we are there, we will need to find the following permissions and enable them for all user roles:



Permission	anonymous user	authenticated user
node_service module		
load any node data	☑	☑
load own node data	☑	☑
services module		
access services	☑	☑
administer services	□	□

For Drupal 5, these permissions are labelled different than they are for Drupal 6. If you are using Drupal 5, then you will need to make sure you enable the **load raw node data** and **access services** permissions.

This is a necessary step that will allow outside applications to make service calls without having to log in to our Drupal system as a valid user (we will cover this in later chapters). It is also important to remember to click the **Save User Permissions** button at the bottom of the page when we are finished checking all the necessary permissions.

We are now done with the boring setup and configuration of Drupal. Now, on to the fun part…building the Flash application to say "Hello World!".

Building a web service-driven "Hello World" application in Flash

As you may have guessed, there is a more-than-typical amount of effort required to create a "Hello World" application that utilizes web services. Many books that teach Flash techniques rarely, if at all, mention the necessary steps required to populate a TextField or MovieClip with data from a remote location. But if you really think about it, this type of architecture is ideal for any scalable or dynamic application. This approach allows for a **Model-View-Controller** (**MVC)** architecture for our Flash applications, which is considered ideal since it separates the User Interface (View) from the Data Set (Model) and Business Logic (Component) of our application. Using web services to populate all of our Flash fields, allows for the component abstraction necessary for an MVC architecture, where Flash suddenly fulfils the role of the View (User Interface) and Drupal fulfils the role as the Model and Component. This enables us to have a truly scalable and manageable web solution.

So, let's take a step-by-step approach on how to accomplish this on the Flash side, which as far as I am concerned, is the fun side!

Step 1: Creating our Flash application

Before we begin this section, we first need to create a new folder to hold all of our changes within this chapter. We can do this by copying the chapter1 folder that we created in the previous chapter, and then paste that new folder and its contents within a new folder called chapter2. Once we have done that, we can rename the chapter1.fla file to chapter2.fla, and then open up that project within our Flash IDE.

With our chapter2 project open, we can shift our focus to the Actions panel within the Flash IDE. Although working with the Actions panel is great for small applications, we will eventually build onto this Flash application, which might make it impractical to keep all of our ActionScript code within the Actions panel. Because of this, we will first need to create a separate ActionScript file that will serve as our main entry point for our Flash application. This will allow us to easily expand our application and add to the functionality without modifying the Actions panel for every addition we make.

Step 2: Creating a main.as ActionScript file

For this step, we will simply create an empty file next to our chapter2.fla file called main.as. After you have created this new file, we will then need to reference it within our Actions panel. To do this, we will use the include keyword in ActionScript to include this file as the main entry point for our application. So, shifting our focus back to the chapter2.fla file, we will then place the following code within the Actions panel:

```
include "main.as";
stop();
```

Now that we are referencing the main.as file for any of the ActionScript functionality, we will no longer need to worry about the Actions panel and add any new functionality directly to the main.as file.

Now, for the following sections, we will use this main.as file to place all of our ActionScript code that will connect and extract information from our Drupal system, and then populate that information in a TextField that we will create later. So, let's jump right in and write some code that connects us with our Drupal system.

Step 3: Connecting to Drupal

For this step, we will first need to open up our empty `main.as` file so that we can add custom functionality to our Flash application. With this file open in our Flash IDE, our first task will be to connect with Drupal. Connecting to Drupal will require us to make a remote call to our Drupal installation, and then handle its response correctly. This will require the use of asynchronous programming techniques discussed earlier in this chapter, along with some standard remoting classes built into the ActionScript 3 library. Since we have already discussed asynchronous programming techniques, I will spend some time here discussing the class used by ActionScript 3 to achieve remote communication. This class is called `NetConnection`.

Using the NetConnection class

The `NetConnection` class in ActionScript 3 is specifically used to achieve remote procedure calls within a Flash application. Luckily, this class is pretty straight forward and does not have a huge learning curve on understanding how to utilize it for communicating with Drupal. Using this class requires that we first create an instance of this class as an object, and then initialize that object with the proper settings for our communication. But let's tackle the creation first, which will look something like this in our `main.as` file:

```
// Declare our Drupal connection
var drupal:NetConnection = new NetConnection();
```

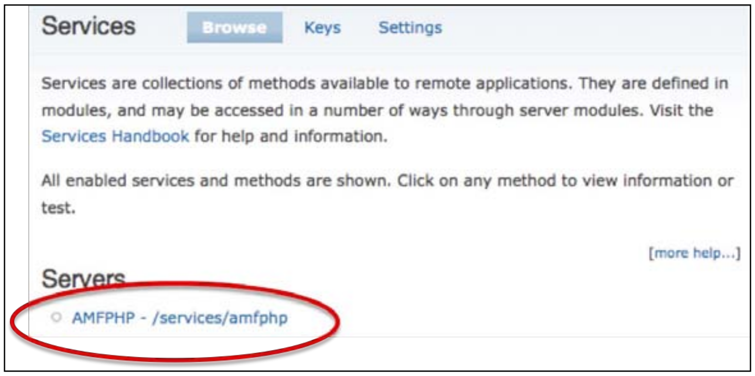
Now, you probably noticed that I decided to name my instance of this net connection `drupal`. The reason for this is to make it very clear that any place in our Flash application where we would like to interact with Drupal, we will do so by simply using our `drupal` `NetConnection` object. But before we use this connection, we must first specify what type of connection we will be using. In any `NetConnection` object, we can do this by providing a value for the variable `objectEncoding`. This variable lets the connection know how to structure the XML format when communicating back and forth between Flash and Drupal. Currently, there are only two types of encoding to choose from: AMF0 or AMF3. AMF0 is used for ActionScript versions less than 3, while AMF3 is used for ActionScript 3. ActionScript 1 and 2 are much less efficient than version 3, so it is highly recommended to use ActionScript 3 over 1 or 2. Since we are using ActionScript 3, we will need to use the AMF3 format, and we can provide this as follows:

```
// Declare our Drupal connection
var drupal:NetConnection = new NetConnection();
drupal.objectEncoding = ObjectEncoding.AMF3;
```

Now that we have an instance ready to go, our first task will be to connect to the Drupal gateway that we set up in the previous section.

Connecting to a remote gateway

Connecting to a remote gateway can be performed using the `connect` command on our `drupal NetConnection` object. But in order for us to connect, we must first determine the correct gateway URL to pass to this function. We can find this by going back to our Drupal installation and navigating to **Administer | Services**. In the **Browse** section, you will see a link to the servers available for remote procedure calls as shown in the following screenshot:



Services are collections of methods available to remote applications. They are defined in modules, and may be accessed in a number of ways through server modules. Visit the Services Handbook for help and information.

All enabled services and methods are shown. Click on any method to view information or test.

[more help...]

Servers

- AMFPHP - /services/amfphp

For every listed server, we can click on each link to verify that the server is ready for communication. Let's do this by clicking on the link for AMFPHP, which should then bring up a page to let us know that our AMFPHP gateway is installed properly. We can also use this page to determine our AMFPHP gateway location, since it is the URL of this page. By observing the path of this page, we can add our AMFPHP server to our `main.as` file by combining the base URL of our site and then adding the AMFPHP services gateway to that base.

```
// Declare our baseURL and gateway string.
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";

// Declare our Drupal connection
var drupal:NetConnection = new NetConnection();
drupal.objectEncoding = ObjectEncoding.AMF3;

// Connect to the Drupal gateway
drupal.connect( gateway );
```

It is important to note that the connect routine is synchronous, which means that once this function is called, we can immediately start using that connection. However, any remote procedure call that we make afterwards, will be asynchronous, and will need to be handled as such. The function that can be used to make these remote procedure calls to Drupal is called `call`.

Using the NetConnection call routine

If you look at the Adobe Help section about the NetConnection call function, you will find the following arguments:

```
function call( command:String,
responder:Responder, … arguments );
```

The first argument, called command, is very easy to understand and use. It is the command that you will send to your remote server to execute. In Drupal, this will be the Service functions that are provided by the System and Node Services that we installed on our server. Since our first task is to simply connect to Drupal, we will first need to use the system.connect command to send to our System Service on our Drupal installation.

The second argument is the responder. This is simply the object that holds the callback functions that are used when the server returns from a remote function call. One callback is used to handle the return value on a successful transfer, while the other callback function is used to handle any error that might have occurred. Since we are programming asynchronously here, we need to first create these two callback functions and then create a new Responder object using both our success and error callback functions. Within our main.as file, we can create the responder with the callback functions as follows:

```
// Declare our baseURL and gateway
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
// Declare our Drupal connection
var drupal:NetConnection = new NetConnection();
drupal.objectEncoding = ObjectEncoding.AMF3;
// Connect to the Drupal gateway
drupal.connect( gateway );
// Set up our responder with the callbacks.
var responder:Responder = new Responder( onConnect, onError);
// Called when Drupal returns with a successful connection.
function onConnect( result:Object )
{
   trace("We are connected!!!");
}
// Called when an error occurs connecting to Drupal.
function onError( error:Object )
{
   for each (var item in error) {
         trace(item);
   }
}
```

The third argument to the `call` routine, and each subsequent argument afterwards, is what will be passed as argument(s) to the remote function that we are calling. You will notice in the function declaration that there are "..." in front of the `arguments` variable. This is called a variable argument function, which means that it can accept any number of arguments into this function. This allows us to provide any number of arguments, which will then be sent as arguments to the remote function that we are calling. In later chapters we will use several different Drupal services where this comes into play.But for the `system.connect` service, there are not many required arguments, so we can just omit this for now.

So, here is the ActionScript code that illustrates how to connect to Drupal using the function `call`.

```
// Declare our baseURL and gateway
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
// Declare our Drupal connection
var drupal:NetConnection = new NetConnection();
drupal.objectEncoding = ObjectEncoding.AMF3;
// Connect to the Drupal gateway
drupal.connect( gateway );
// Set up our responder with the callbacks.
var responder:Responder = new Responder( onConnect, onError);
// Connect to Drupal
drupal.call("system.connect", responder);
// Called when Drupal returns with a successful connection.
function onConnect( result:Object )
{
   trace("We are connected!!!");
}
// Called when an error occurs connecting to Drupal.
function onError( error:Object )
{
   for each (var item in error) {
      trace(item);
   }
}
```

We can now test this out by running our application. Congratulations, we are now connected! Let's move on.

Step 4: Session handling

Now that we are connected to Drupal, our next intuition is for us to dive in and start extracting data from our Drupal site; but this cannot be accomplished without first handling the session ID for our connection with Drupal. The session ID is simply a unique identifier for every connection made with any web site. The session ID allows a web site to keep a track of every single person navigating the site, and therefore, give them certain permissions depending on whether that person is logged in as a user who can perform certain tasks. Each browser then uses cookies to store the session ID of that user so that the next time they open up their browser, their session is restored by setting the session ID to the same value as what was saved in the cookie. We will use this session ID in our application for every call that we make to Drupal to validate our connection. By default, the Services module assumes that any application making calls to the Drupal system is an "anonymous" application, and therefore, does not allow that application to perform specific tasks. By utilizing the session ID, we are allowing our Flash application to validate itself with our Drupal installation so that the typical user management system that Drupal employs to access content is utilized. With that said, let's modify our previous code to handle the session ID for our connection with Drupal.

Connecting to Drupal using system.connect

Although session handling may sound complicated, the effort involved is minor since the ID is returned to us after the Drupal connection has been made using the `system.connect` call. And since we will be using this session ID for other routines, we need to declare this variable in the global realm of our ActionScript code (that is, not within a function). So, the modified code for session handling will look like the following:

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";

// Declare our Drupal connection
var drupal:NetConnection = new NetConnection();
drupal.objectEncoding = ObjectEncoding.AMF3;

// Connect to the Drupal gateway
drupal.connect( gateway );

// Set up our responder with the callbacks.
var responder:Responder = new Responder( onConnect, onError);

// Connect to Drupal
drupal.call("system.connect", responder);
```

```
// Called when Drupal returns with a successful connection.
function onConnect( result:Object )
{
   // Set our sessionId variable.
   sessionId = result.sessid;

   trace("We are connected!!!");
   trace("Session Id: " + sessionId);
}

// Called when an error occurs connecting to Drupal.
function onError( error:Object )
{
   for each (var item in error) {
      trace(item);
   }
}
```

We can now verify that this works by running our application. We should see, within the output panel, our connection with Drupal followed with a valid session ID. We are now ready to move on.

> If you are not using a localhost for your server, you will most likely see a Security dialog box when you run the application for the first time. If this occurs, you will need to add the compiled SWF file to the **Global Security Panel** by navigating to `http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager04.html`. Once you are on this webpage, you can add the compiled SWF file by clicking on the **Edit Locations** drop-down box, then select **Add Location**. This will bring up a new dialog where you can click on **Browse for files** button and choose the SWF file within our `Chapter 2` directory.

Step 5: Drupal says "Hello World"

Now, we are getting to the fun part… loading Drupal data into our Flash application. But before we can write the ActionScript to load node data from Drupal, we must first revisit our Drupal web site and create a new node that we will use to say "Hello". Moving back to our Drupal web site, let's create a new node by going to **Create Content | Page**. The page title is what we will use to say hello, so where it asks for a title, put in the text "Hello World!", and then go down to the very bottom of the page and hit the **Submit** button. We should then see our new page created that says, "Hello World!" for the title. It is very important for us to also remember the node ID of the node that we just created. We can do this by simply looking at the URL in our browser and writing down the number that comes after **http://locahost/drupal6/node/**. We will use this number to load our node in Flash, so just make sure you either remember it or write it down.

Loading a node in Flash

Now that our node is ready, we can write the code in ActionScript that loads this node and then parses out the title. This is how we are going to build our "Hello World" application using Flash and Drupal. So, let's move back to our Flash IDE and take a look at the code that we have so far.

Picking up where we left off, we are finally at the point where we have successfully connected to our Drupal web site and received the session ID. The next step in this process is to get the node information from the node that we just created in our Drupal site. To accomplish this, we will be using the Service method called `node.get`, which takes two arguments: the session ID, and the node ID of the node we wish to load. This is where our variable arguments come into play, since we have two arguments that we need to provide.

At this point, it is considered best practice to create a separate function that combines the functionality of creating our responder with the Drupal service call to load a node. For the sake of simplicity, we will call this new function `loadNode`, where it will take a single argument (the node ID), and then use that argument and pass it along to the `node.get` service function. The Drupal node service will then return the node object for the node that we are requesting by calling the `callback` function that we provided within the responder. Within each node object, we will have access to the Title, Body, and any other fields of data that are associated with a Drupal node. Since we used the node title to say "Hello World", we can create an `onNodeLoad` function to print out the title field for the node object returned from Drupal. Each of these functions will look as follows:

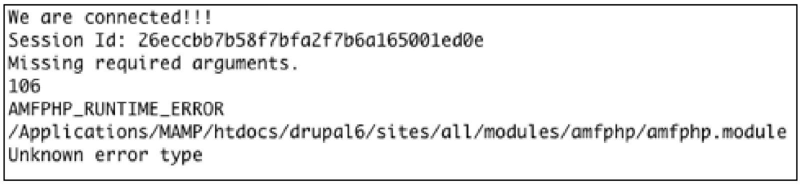
```
// Connect to Drupal
drupal.call("system.connect", responder);

// Loads a Drupal node.
function loadNode( nid:Number )
{
   // Set up our responder with the callbacks.
   var nodeResponse:Responder = new Responder( onNodeLoad, onError);

   // Call Drupal to get the node.
    drupal.call( "node.get", nodeResponse, sessionId, nid );
}

// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   trace( node.title );
}
```

We can now use the loadNode function to load any Drupal node by passing in the node ID for the node we wish to load. For example, if we wish to load a node with an ID of 2, we can simply call loadNode(2), and all the complex functionality is now abstracted within that single function.

Now for the big question…where do we place the call to loadNode so that we can load our "Hello World!" node that we just created? This seems simple enough, but leads to a significant gotcha where asynchronous software becomes a little confusing. To illustrate, let's take a look at the following code where I am attempting to connect to Drupal and then make a simple call to loadNode directly afterwards. Many developers who just start out using web services very often attempt to do the following and get very frustrated when they discover that it just does not work.

```
// Connect to Drupal
drupal.call("system.connect", responder);

// Load our "Hello World" node (ID = 2)
loadNode( 2 );
```

If we were to run this application, we will quickly see an issue by looking at the Output panel. This panel prints out the response returned from our Services module after we make our call to load a node. Since an error occurred, our onError function is called, and then the following error is printed out:



```
We are connected!!!
Session Id: 26eccbb7b58f7bfa2f7b6a165001ed0e
Missing required arguments.
106
AMFPHP_RUNTIME_ERROR
/Applications/MAMP/htdocs/drupal6/sites/all/modules/amfphp/amfphp.module
Unknown error type
```

This error is the result of an elusive software bug called race condition.

Programming without race conditions

If we take another look at the previous code, we will see that we are making a call to loadNode directly after the system.connect function call. In the world of synchronous software, this would work just fine, but since we are dealing with web service communication, our function calls to Drupal are not returned after each call is made. Instead, each call that we make to Drupal can take any amount of time before the result is returned using a callback function. To the trained eye, this is obvious, but for the developers just learning asynchronous software behaviour, this can be quite the head scratcher. Taking asynchronous software interaction into account, we can now determine that our error occurred because we were making the call to get the node information *before* we received any indication that we have successfully connected, from the system.connect command. There is no guarantee that a response from the server will make it back in time before we end up making the call to loadNode. In asynchronous programming, this is most commonly referred to as a race condition, where you are betting that the return from system.connect will beat the call to loadNode. Unfortunately, you will most likely lose this bet, and the result will be a strange error that does nothing to expose the smoking gun.

> When describing the concepts behind *race conditions*, I always like to think of Drupal as having a bad case of selective hearing. Since I too have selective hearing, I can always imagine the communication between Flash and Drupal much like how my wife communicates with me.
>
> Over the years, she has learned the hard way that in order to give me a command successfully, such as takeOutGarbage, she must first get my attention by calling my heyTravis! function first and then wait for my yesDear? response. If she does not wait for my response before issuing her command, I, almost always, misunderstand her and do something completely different. For example, if she were to call my heyTravis! function followed directly with two commands: changeBabyDiaper and takeOutGarbage, there would be a good chance that I would end up throwing the baby out with the garbage.
>
> Taking this into account, we should always get Drupal's attention first using the system.connect message, and then, we must wait to get a response before issuing any additional commands.

Programming with race conditions is considered very poor programming practice, and will most often lead to an extremely elusive software bug in your application. In fact, I can easily say that with all my experience debugging software in complex applications, it is always the race condition that is the hardest bug to find and correct. So, let's take a moment and learn how to modify the previous code so that it will never hit a race condition.

The trick is to simply move the call to loadNode after we receive notification from Drupal that we are connected and have a valid session ID. After we look at the following code, it will seem very obvious, but you would be surprised how often this gotcha seems to crop up in complex software applications. The modified code should look like the following (assuming the node ID you created was 2):

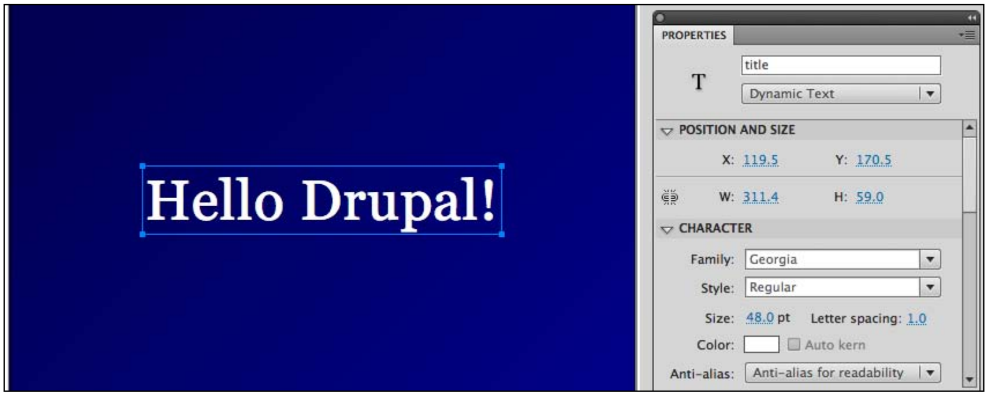
```
// Connect to the Drupal gateway
drupal.connect( gateway );

// Set up our responder with the callbacks.
var responder:Responder = new Responder( onConnect, onError);

// Connect to Drupal
drupal.call("system.connect", responder);

...
```

```
...
// Called when Drupal returns with a successful connection.
function onConnect( result:Object )
{
   // Set our sessionId variable.
   sessionId = result.sessid;

   trace("We are connected!!!");
   trace("Session Id: " + sessionId);

   // Load our "Hello World" node (ID = 2)
   loadNode( 2 );
}
```

So, after we run this application, we should get a very nice surprise… a "Hello World" from Drupal! But we are still not done here; our next step is to hook up the text in our Flash TextField to show this exciting text.

Step 6: Hooking up the text

In this next step, we will open up our `chapter2.fla` project file, where we will give our TextField an instance name so that it can be referenced within ActionScript. Fortunately, this step is very simple and only requires that we select the TextField, and then give it an instance of **title** in the **PROPERTIES** panel as shown in the following screenshot:



Hello Drupal!

PROPERTIES

T title

Dynamic Text

POSITION AND SIZE

X: 119.5 Y: 170.5

W: 311.4 H: 59.0

CHARACTER

Family: Georgia

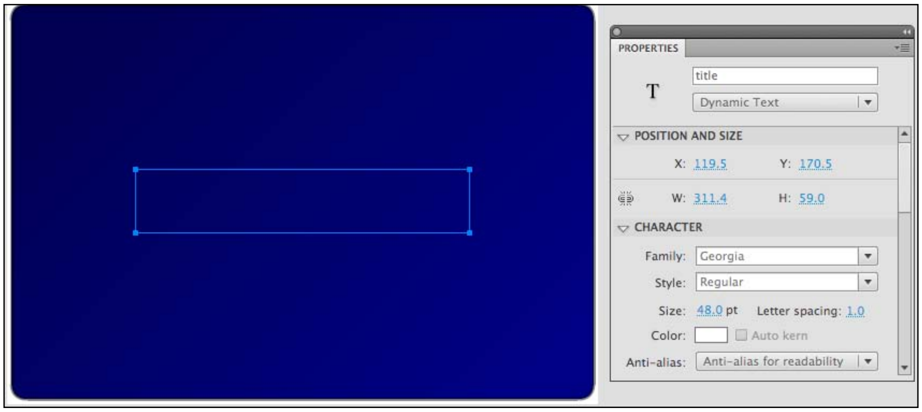
Style: Regular

Size: 48.0 pt Letter spacing: 1.0

Color: □ Auto kern

Anti-alias: Anti-alias for readability

Now that we have given our TextField an instance, the next step is to remove the text "Hello Drupal" from this TextField so that we can determine if Drupal node data is used instead.



PROPERTIES

title

Dynamic Text

POSITION AND SIZE

X: 119.5 Y: 170.5

W: 311.4 H: 59.0

CHARACTER

Family: Georgia

Style: Regular

Size: 48.0 pt Letter spacing: 1.0

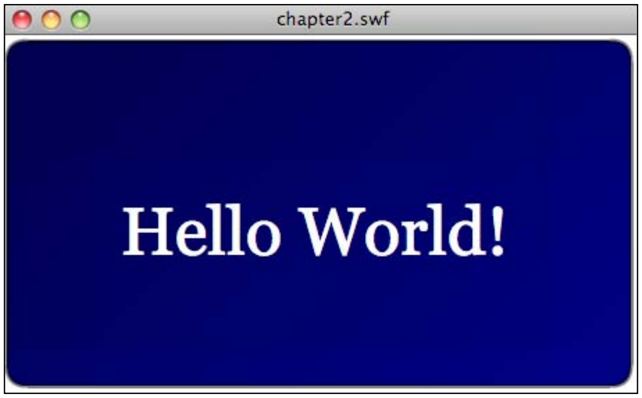
Color: □ Auto kern

Anti-alias: Anti-alias for readability

We can now shift our focus back to the `main.as` file, where we will change our trace statement within our `onNodeLoad` function so that it sets the text of this TextField instead of just printing it to the Output panel. We can do this in ActionScript by using the following code:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
}
```

We can now run our application and see our TextField show the title for our **Hello World!** node.



chapter2.swf

Hello World!

Now that our application is starting to look like a real Flash application, we need to take some extra steps to make sure that it is flexible by allowing any node ID to be used to say "Hello World".

Step 7: Passing the node ID using FlashVars

Since the goal of our Flash application is to dynamically load Drupal node information, we will need a way to tell our Flash application which node to load. We can easily hard code the node ID of our "Hello World" Drupal node, but this approach does not give us much flexibility to apply it to any node within our Drupal system. We can solve this issue by utilizing FlashVars to pass the node ID to our Flash application.

Using FlashVars in a Flash application

Flash variables (or FlashVars) are special variables that are passed to a Flash application that are used to provide a specific functionality for a common application. They are passed to the Flash application when it is embedded within an HTML page using the `<object>` element. For example, we can tell our Hello World application to load the node data from node 2 using the following HTML code:

```
<object width="320" height="240">
   <param name="movie" value="helloworld.swf" />
   <param name="wmode" value="transparent" />
   <param name="allowfullscreen" value="true" />
   <param name="FlashVars" value="node=2" />
   <param name="quality" value="high" />
</object>
```

Within ActionScript, the `node` variable is then passed to the `root` structure and can be referenced within our `main.as` file using the `root.loaderInfo.parameters` construct. For example, we can determine the node ID passed to our Flash application by using the following code within our `main.as` file:

```
root.loaderInfo.parameters.node
```

Using this information, we can now create a `nodeID` variable at the top of our `main.as` file, where we will set it to the node ID passed to our Flash application. We can then replace our hard-coded node value with this variable so that our Flash application is no longer dependent on a specific node value.

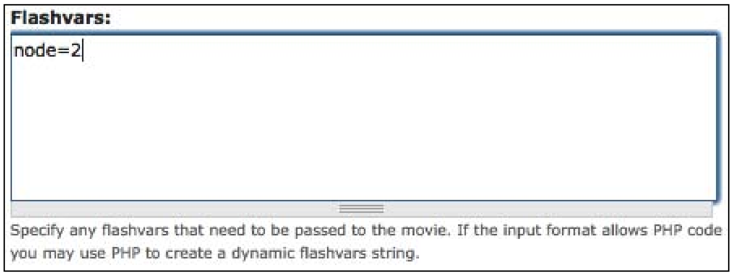
```
// Declare our variables
var baseURL:String = "http://localhost/drupalbook";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";
var nodeId:Number = root.loaderInfo.parameters.node;
```

```
...
...
// Called when Drupal returns with a successful connection.
function onConnect( result:Object )
{
   // Set our sessionId variable.
   sessionId = result.sessid;

   trace("We are connected!!!");
   trace("Session Id: " + sessionId);

   // Load our node.
   loadNode( nodeId );
}
```

We are now finished with our "Hello World" application. At this point, we can run our Flash application so that it will create the `chapter2.swf` file, which we will then use to add to our Drupal web site.

Step 8: Adding it to Drupal

In this step, we can go back to the lessons learned from the previous chapter where we used the FlashNode module to add a SWF file to our Drupal website. To start, we will create a new Flash node by navigating to **Create Content | Flash**, and then give it a title of "Hello World Application". After we have done that, we can then do what we did in the previous chapter and select our newly created SWF file using the **Flash file** input field. The next step, however, will require us to pass in the FlashVar so that we can tell our application which node to load. To do this, we will simply expand the **Advanced flash node options**, and place **node=2** within the **FlashVars** text field.

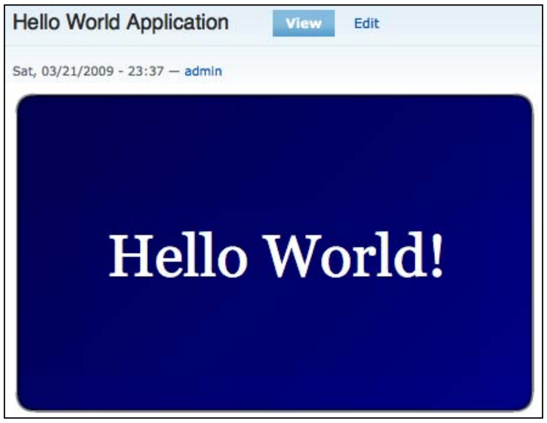


Flashvars:

node=2

Specify any flashvars that need to be passed to the movie. If the input format allows PHP code you may use PHP to create a dynamic flashvars string.

We can now save our new node, and see our new dynamic Flash application in action.



And congratulations, we have just said "Hello World" using both Flash and Drupal!

Summary

In this chapter we covered a lot of ground, learning all of the concepts that govern web service interaction between Flash and Drupal. When dealing with web services, it is very important to understand how two remote applications communicate and how to develop our software to account for its asynchronous interaction. This is typically always overlooked when developers create their first Flash applications for Drupal, and can easily be avoided if the concepts of web service interaction are understood and taken into account. Each key concept is highlighted as follows:

- Flash and Drupal communicate asynchronously. This means that each function call made to Drupal from Flash does not return the result immediately after the call was made. Instead, we need to utilize a `callback` function that is triggered when Drupal returns the result from our function call.
- We need to wait until the Flash application has finished connecting to Drupal before making any other calls. Otherwise, we will have a race condition where our `sessionID` is invalid for our other calls.

In the next chapter, we will take the concepts learned within this chapter and take it to the next level by using the popular CCK module to expand the amount of content that we can utilize within our Flash applications.

3
Flash and CCK

We have built a "Hello World" application, and understood how Flash extracts data from our Drupal installation; our next task will be addition of content in our Flash application using Drupal's **Content Construction Kit**, also known as **CCK**.

The CCK module allows you to add custom content types and data fields to your Drupal web site, which can then be utilized to create a more customized Flash application. In this chapter, we will go over all the technical aspects of Flash and CCK by walking through a real-life example of building a Flash widget that shows recipes for a cooking web site. Here is what we will cover in this chapter:

- Overview of a typical recipe web site
- Installing and understanding Drupal's Content Construction Kit
- Creating a custom Recipe content type for your Flash application
- Adding custom fields to the Recipe content type
- Building a Flash application that uses data from CCK
- Adding ScrollBars to Flash CCK TextFields
- Creating a Drupal node template for Flash

Overview of a typical recipe web site

In this chapter our goal is to create a Flash widget that shows each recipe for a cooking web site. Since the structure of most recipe web sites is fairly consistent, I believe this example will easily illustrate how to create any form of custom content and display it within a Flash application. I am confident that after you read this chapter, you should be able to easily apply the lessons learned in this chapter to any type of custom implementation that your web site requires. So, let's take a moment to set up Drupal so that its data structure resembles that of a common recipe web site.

If you were to visit any popular recipe web site, you will most likely see a categorized listing of recipes ranging from cake to pot roast. When you click on one of these listings, you will see a new page that gives the information necessary to make that recipe. In Drupal terms, this recipe page is a single node that is categorized using a content type called Recipe.

> It is important to note that in Drupal, the terms **node type** and **content type** are used interchangeably. Although, I will almost always use the term content type in this book, there are many online tutorials and documentation that use the term node type to refer to different types of content. Just keep in mind that node type and content type are both used to refer to the same thing.

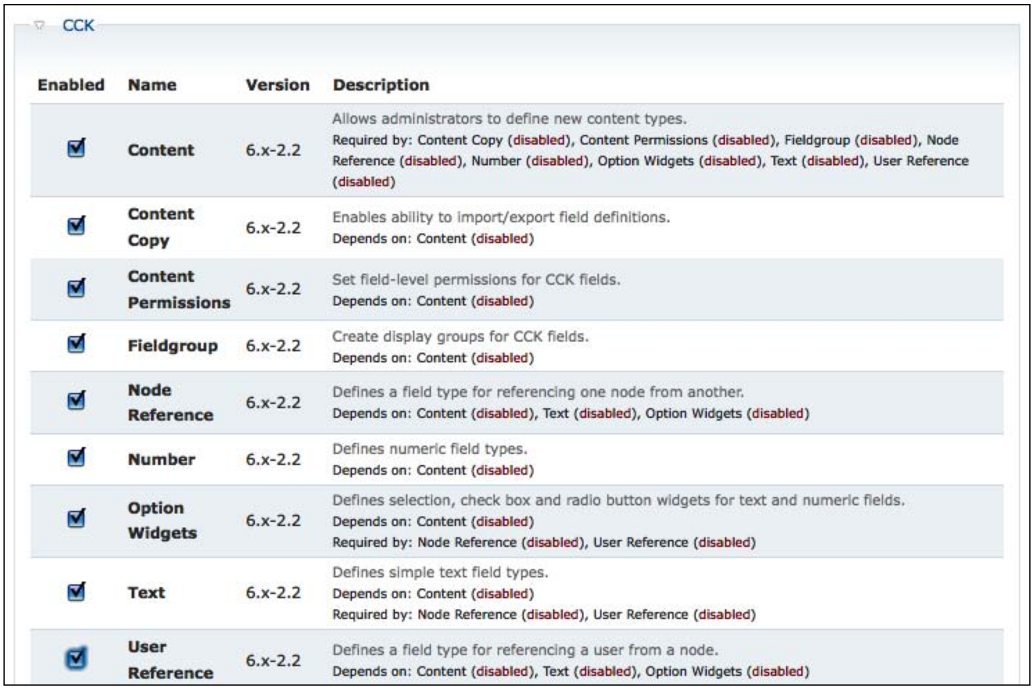
Our goal in this chapter will be to replace the HTML page for each Recipe node with our own Flash application. We will set up a simple Recipe content type in Drupal with some necessary fields, which will then be displayed in our Flash application. Having said that, each recipe in our cooking web site should have the following information:

- Title
- Description
- Ingredients
- Instructions

But before we can dive into Flash, our first task will be to use CCK to build this structure from within the Drupal CMS.

Using Drupal's Content Construction Kit

The CCK module is one of Drupal's most popular, contributed modules, and for a good reason. It gives the administrator the ability to create custom content types as well as custom fields that are essential for creating a web site that manages specific forms of content. We will use this module to create the structure required to enter content for each recipe in our Drupal web site. But before we can start adding custom content to our Drupal installation, it is important to first download and install the Content Construction Kit from `http://www.drupal.org/project/cck`. Once we have this module within our modules folder, we can then navigate to **Administer | Modules**, where we will select all of the following modules within CCK:



Enabled	Name	Version	Description
☑	Content	6.x-2.2	Allows administrators to define new content types. Required by: Content Copy (disabled), Content Permissions (disabled), Fieldgroup (disabled), Node Reference (disabled), Number (disabled), Option Widgets (disabled), Text (disabled), User Reference (disabled)
☑	Content Copy	6.x-2.2	Enables ability to import/export field definitions. Depends on: Content (disabled)
☑	Content Permissions	6.x-2.2	Set field-level permissions for CCK fields. Depends on: Content (disabled)
☑	Fieldgroup	6.x-2.2	Create display groups for CCK fields. Depends on: Content (disabled)
☑	Node Reference	6.x-2.2	Defines a field type for referencing one node from another. Depends on: Content (disabled), Text (disabled), Option Widgets (disabled)
☑	Number	6.x-2.2	Defines numeric field types. Depends on: Content (disabled)
☑	Option Widgets	6.x-2.2	Defines selection, check box and radio button widgets for text and numeric fields. Depends on: Content (disabled) Required by: Node Reference (disabled), User Reference (disabled)
☑	Text	6.x-2.2	Defines simple text field types. Depends on: Content (disabled) Required by: Node Reference (disabled), User Reference (disabled)
☑	User Reference	6.x-2.2	Defines a field type for referencing a user from a node. Depends on: Content (disabled), Text (disabled), Option Widgets (disabled)

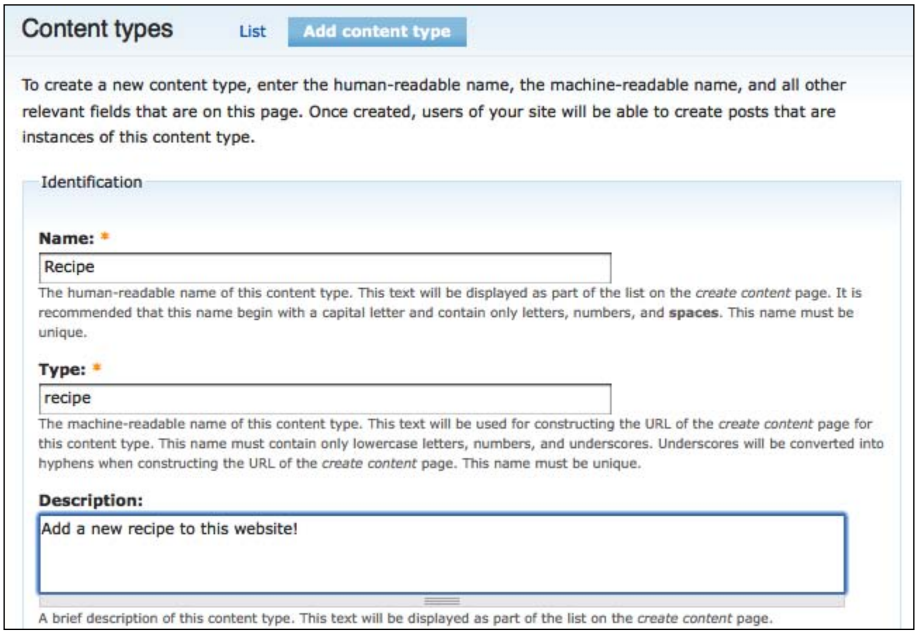
Once we have all of the CCK modules selected, we can then click on the **Save Configuration** button at the bottom of the page to install CCK into our web site. Now that we have CCK installed, let's create our Recipe content type.

Creating a new content type

Our first step in building the Drupal structure for a recipe web site will be to create the content type that will hold each recipe. By default, Drupal comes equipped with only two different types of content: Page and Story. Each one of these content types is placed in the default installation, simply because they provide a very generic method for adding content on our Drupal site. When we add a new Page and Story to our web site, we are asked to provide a Title and Body for that piece of content. These are then used to construct a web page, where the title of the web page is provided from the Title field; and the content of that page is provided using the Body field. A recipe, however, requires a specific set of data that will be used to describe the "content" that makes up that recipe. For example, the Body field really does not make much sense when related to recipes since this is very general. A better design will be having several fields that relate specifically to a recipe. These fields could be: Ingredients, Instructions, Description, and so on. CCK gives us the ability to create a custom content type called Recipe, where we can then provide custom fields to describe that piece of content.

First, let's create our custom Recipe content type by walking through the following steps:

1. Navigate to the **Drupal Administrator** section by clicking on the navigation link called **Administer**.
2. Once you are in the administrator section, you will need to click on the link called **Content Types** in the Content Management section.
3. Now, click on the link that says **Add content Type**.
4. This should then bring up a page with open fields for you to enter information about the content type that you wish to create. Each of these fields are described as follows:
   - **Name**—This is the name of your content type that will be visible to any visitor(s) of this site.
   - **Type**—This is the internal name for your content type. This name should not contain any spaces or special characters, and it is also considered best practice to make this name all lowercase.
   - **Description**—This is the text that is presented below the **Name** of this content type when our visitors wish to create a new piece of content. It is simply used to better describe what the **Name** means.
5. We can now fill out all the information for our new piece of content as shown in the following screenshot:



Content types

To create a new content type, enter the human-readable name, the machine-readable name, and all other relevant fields that are on this page. Once created, users of your site will be able to create posts that are instances of this content type.

Identification

Name: *
Recipe

The human-readable name of this content type. This text will be displayed as part of the list on the *create content* page. It is recommended that this name begin with a capital letter and contain only letters, numbers, and **spaces**. This name must be unique.

Type: *
recipe

The machine-readable name of this content type. This text will be used for constructing the URL of the *create content* page for this content type. This name must contain only lowercase letters, numbers, and underscores. Underscores will be converted into hyphens when constructing the URL of the *create content* page. This name must be unique.

Description:
Add a new recipe to this website!

A brief description of this content type. This text will be displayed as part of the list on the *create content* page.

- Let's save this new content type by clicking on the **Save content Type** button at the bottom of the page.

We now have a new content type that will be used for each individual recipe on our web site. We can test this out by clicking on the link in the Navigation menu that says **Create Content**, and then clicking on the link that says **Recipe**. This should bring up a new page that asks us to enter the Title and Body for that Recipe. If our users were asked to do the same, they would most likely be very confused at this point. The input for Title would most likely be understood, but as mentioned before, the term Body does not mean anything when submitting a recipe and would typically lead to inconsistent content, based on the assumptions of what the term Body means to a recipe. The solution to this problem is to create specific fields that describe each recipe in our system, and once again, CCK comes to the rescue.

Adding custom fields to your Recipe content type

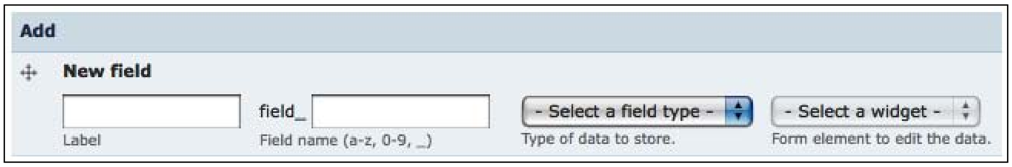
In this section you will understand that the true power of CCK does not come from its ability to add new content types to your Drupal system, but in its ability to add customized fields to each new content type. And the process of adding new fields is surprisingly simple and intuitive.

To add a new field to the Recipe content type, we will need to revisit the **Content Types** page by clicking on **Administer | Content Types**.

1. Once you are there, you will see a listing of all the content types in the system along with the Recipe content type. Beside each content type are a set of links where you can edit, delete, or manage the fields of each individual content type. To add fields to each recipe, we will first click on the **manage fields** link next to the Recipe content type.
2. This will bring up a listing of all the fields that are associated with our Recipe content type, and also give us the ability to add, delete, reorder, or edit each individual field. However, for our recipe, we want to add some new fields that will be more descriptive of each recipe that is created.

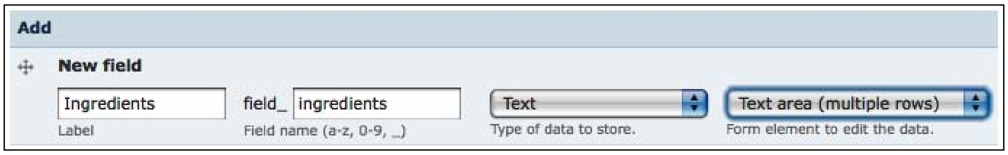
Adding a new field

Within the **Add | New field** section of the **Manage fields** page, we are given the ability to add a new field to our Recipe content type. Each input box in this section is described as follows:



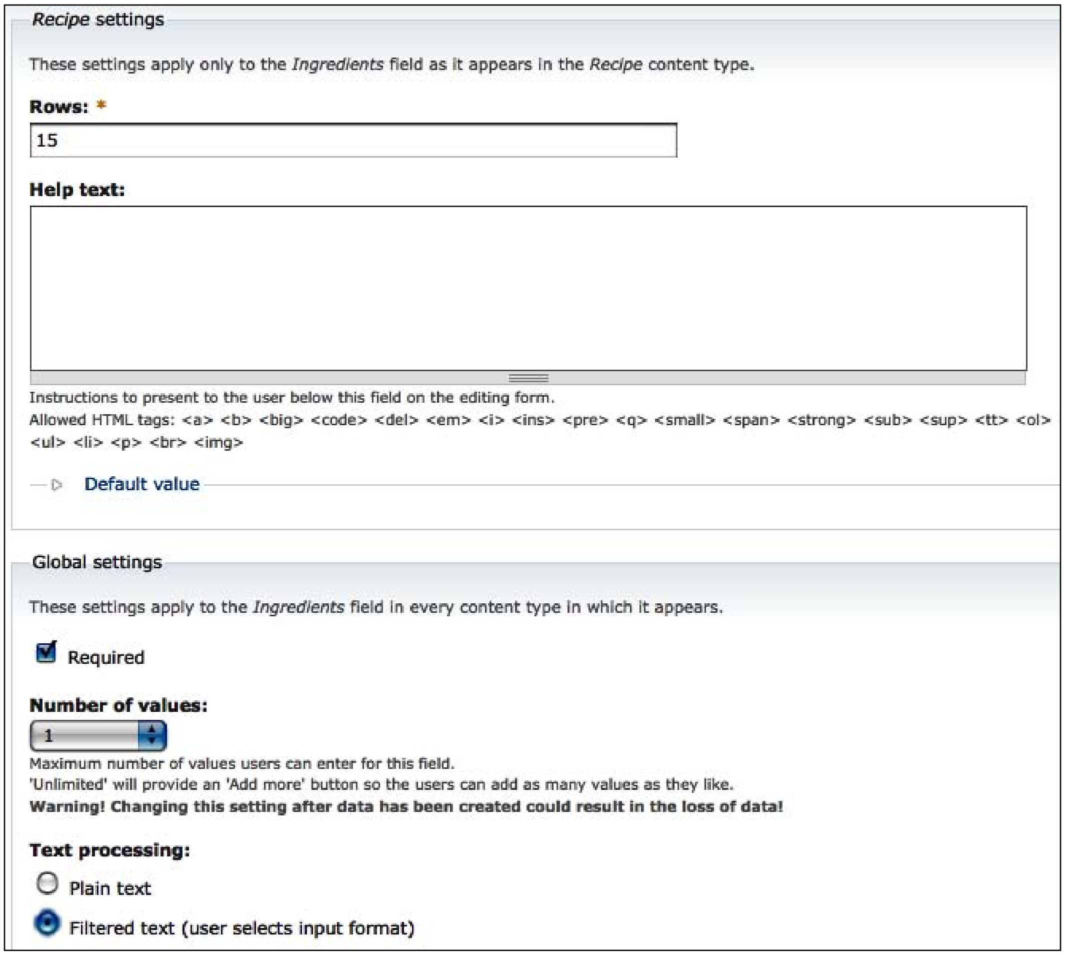
- **Label**—This is the human readable name for this field, such as **Ingredients**.
- **Field name**—This is the machine name for this field. A machine name is used for internal identification and will most likely not be seen by any common user. However, this name is extremely important when writing software that references this particular field; so, it is still important to give it an easily distinguishable name. It must also be all lowercase and not include any spaces, such as **ingredients**.
- **Type of data to store**—This is the type of data for the new field, such as decimal, integer, text, and so on.
- **Form element**—This allows for each type of data to be entered in different ways. For example, you can enter a number by either typing it into a text field or by selecting if from a drop-down box. Both methods store a data type of integer, but they are entered differently.

The first field that we will add to our Recipe content type is the **Ingredients** field. Before we start adding the necessary data in the required fields, let's first think about how we would like our users to input the ingredients for their recipe. Most likely, they would need to enter the **Ingredients** inside of a large text area that allows for more than one line. From a CCK perspective, this would simply be the equivalent of us adding a **Text** field that allows for multiple rows of data entry. Using that information, we can easily submit the correct data for our new field as shown:


Label: Ingredients
field_ ingredients
Type of data to store.: Text
Form element to edit the data.: Text area (multiple rows)

Now, to add this field, we will simply click on the **Save** button at the bottom of the **Manage Fields** page. This will then bring you to a separate page that allows you to further refine the behavior of our new field.

I won't go over all the elements on this page because it is very self-explanatory, but we want to change the number of rows for our **Text area** to be more than 5, since recipe ingredients will almost always take up more than 5 lines. To do this, simply change the **Rows** text box to have 15 lines. We also want to make sure that the user always enters the ingredients for every recipe that they submit. We can enforce that by simply clicking on the **Required** checkbox. We also need to check the **Filtered text** radio button since this will auto-format the input from the user. All of our settings should now look like the following:



Recipe settings

These settings apply only to the *Ingredients* field as it appears in the *Recipe* content type.

Rows: *

15

Help text:

Instructions to present to the user below this field on the editing form.
Allowed HTML tags: <a> <b> <big> <code> <del> <em> <i> <ins> <pre> <q> <small> <span> <strong> <sub> <sup> <tt> <ol> <ul> <li> <p>
 <img>

▷ Default value

Global settings

These settings apply to the *Ingredients* field in every content type in which it appears.

☑ Required

Number of values:

1

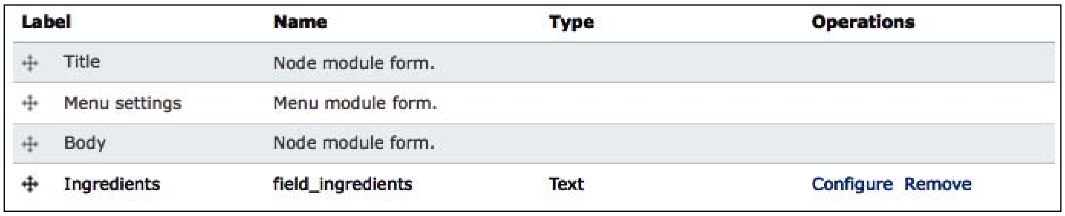
Maximum number of values users can enter for this field.
'Unlimited' will provide an 'Add more' button so the users can add as many values as they like.
Warning! Changing this setting after data has been created could result in the loss of data!

Text processing:

○ Plain text

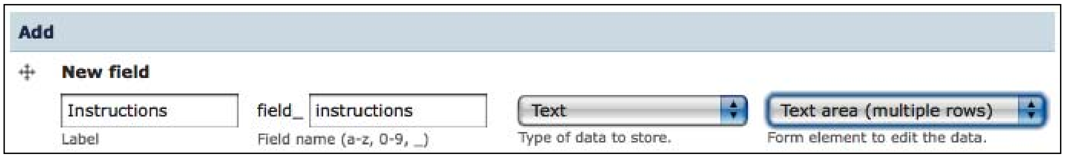
● Filtered text (user selects input format)

After we are done with setting up our field, we can save it by clicking on the button that says **Save field settings**. Then, we should be taken back to the **Manage fields** page, where our new field has been added to the **Recipe** content type.



Label	Name	Type	Operations
Title	Node module form.		
Menu settings	Menu module form.		
Body	Node module form.		
Ingredients	field_ingredients	Text	Configure Remove

After we have the **Ingredients** field set up for the Recipe content type, our next step is to repeat the steps given previously to create an **Instructions** field for the Recipe content type.

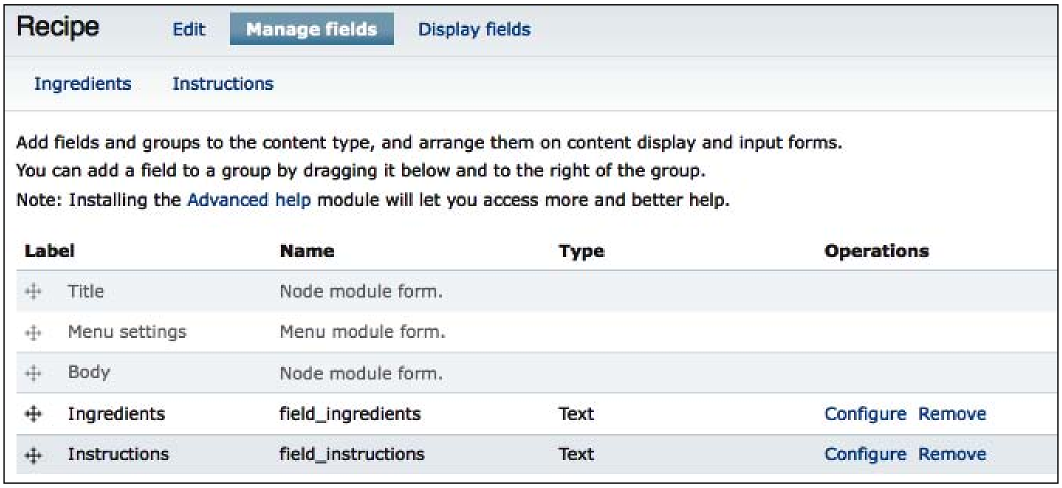


Add

New field

Instructions	field_ instructions	Text	Text area (multiple rows)
Label	Field name (a-z, 0-9, _)	Type of data to store.	Form element to edit the data.

When we are finished walking through the steps above for the **Instructions** field, we should then have the following fields within our **Recipe** content type:



Add fields and groups to the content type, and arrange them on content display and input forms.
You can add a field to a group by dragging it below and to the right of the group.
Note: Installing the Advanced help module will let you access more and better help.

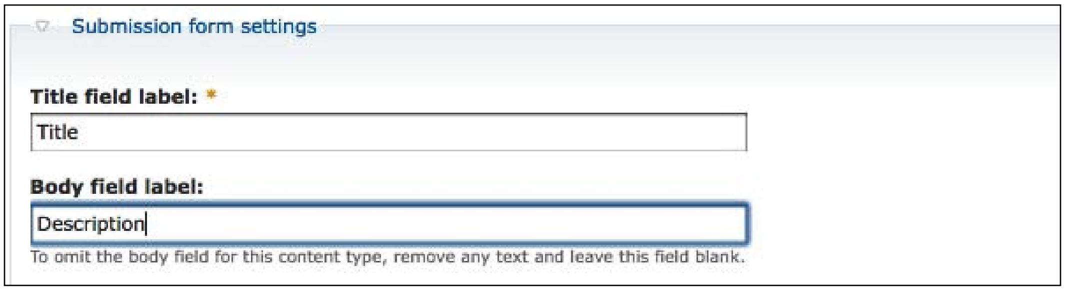
Label	Name	Type	Operations
Title	Node module form.		
Menu settings	Menu module form.		
Body	Node module form.		
Ingredients	field_ingredients	Text	Configure Remove
Instructions	field_instructions	Text	Configure Remove

We should now have a **Recipe** content type that has all the necessary elements for any user to accurately submit a new recipe to our web site. So, let's go ahead and test this out by going to **Create Content | Recipe**. Although, you will see the **Instructions** and **Ingredients** fields in the recipe form, you will most likely also notice that the **Body** field is still there. This is because the **Body** field is a default field for every node in the Drupal system, but luckily, this can also be easily changed.

Changing the default Body field

The difference between default fields and custom fields for a content type may cause some confusion for any Drupal beginner, simply because they are configured in two different places. If you go back to the **Content Types | Manage Fields** section for the **Recipe** node type, you will probably notice that there is no option to configure the **Body** field like there is for the custom types that we created. To get around this, Drupal has given the ability to customize these fields by editing content type using the **Edit** link for that content type.

After we click on this link, we will expand the section that says **Submission Form Settings**. In this section, we should see some options where we can edit the **Title** and **Body** fields for the Recipe content type. Although we can completely remove the Body field by simply deleting the text in the **Body Field label**, we can also change this text so that it represents a recipe. Let's change this field to say **Description**, which will prompt each user to give each submitted recipe a description, which we can use later when we build a page that lists many different recipes.



Title field label: *

Title

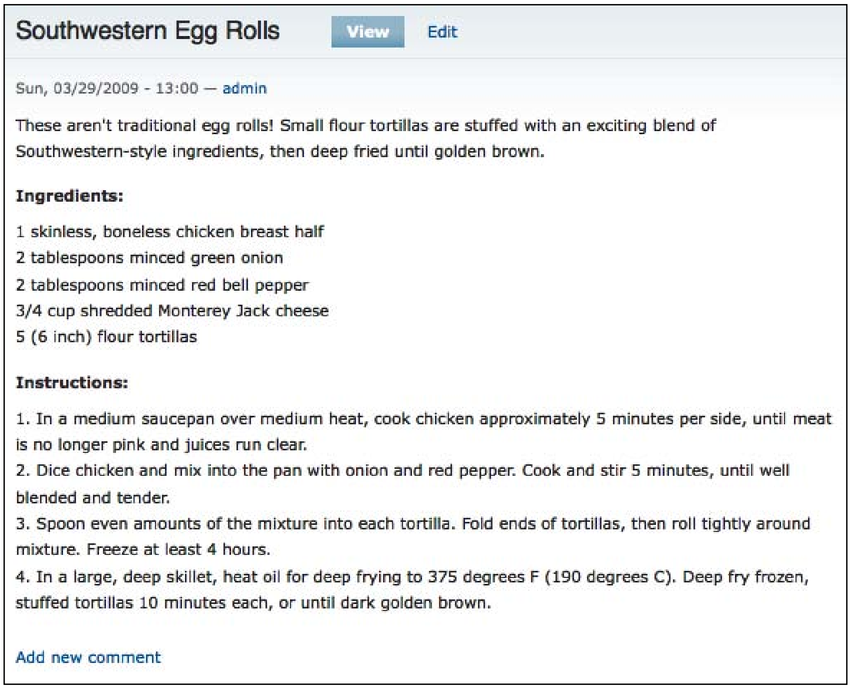
Body field label:

Description

To omit the body field for this content type, remove any text and leave this field blank.

We can save the content type by clicking on the **Save content type** button at the bottom of the page.

Now, when we go back to **Create Content | Recipe**, we will finally see that our recipe submission form resembles what any user would expect when they wish to enter a new recipe. We can take this moment and create a sample recipe by filling out the contents for the **Title**, **Description**, **Ingredients**, and **Instructions** for a new recipe, and then **Save** that recipe when we are finished.



Southwestern Egg Rolls

Sun, 03/29/2009 - 13:00 — admin

These aren't traditional egg rolls! Small flour tortillas are stuffed with an exciting blend of Southwestern-style ingredients, then deep fried until golden brown.

Ingredients:

1 skinless, boneless chicken breast half
2 tablespoons minced green onion
2 tablespoons minced red bell pepper
3/4 cup shredded Monterey Jack cheese
5 (6 inch) flour tortillas

Instructions:

1. In a medium saucepan over medium heat, cook chicken approximately 5 minutes per side, until meat is no longer pink and juices run clear.
2. Dice chicken and mix into the pan with onion and red pepper. Cook and stir 5 minutes, until well blended and tender.
3. Spoon even amounts of the mixture into each tortilla. Fold ends of tortillas, then roll tightly around mixture. Freeze at least 4 hours.
4. In a large, deep skillet, heat oil for deep frying to 375 degrees F (190 degrees C). Deep fry frozen, stuffed tortillas 10 minutes each, or until dark golden brown.

Add new comment

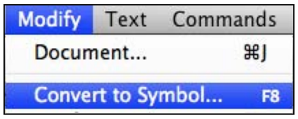
Our next step is to show this newly created node inside our Flash recipe application.

Showing CCK fields in Flash

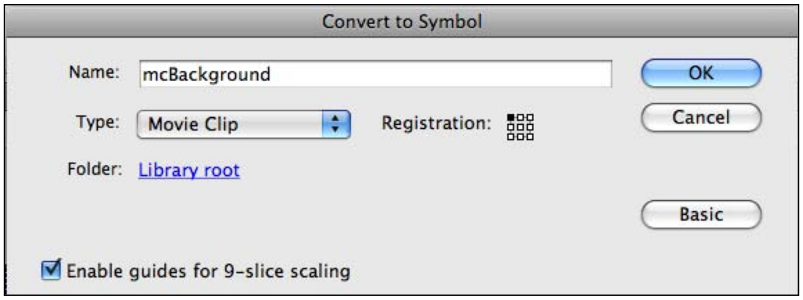
We now shift our focus to Flash and pick up where we left off from the previous chapter. We will not only show the node title, but also show the CCK fields that we just created. We will start by first copying the `chapter2` directory that we created in the previous chapter, and then paste our copy as `chapter3`. We will then rename the `chapter2.fla` project within this folder as `chapter3.fla`. Once we have our new project for this chapter, we can open it up, where we can modify it to include the new fields that we just created.

Building a Recipe widget in Flash

Now that we have our Chapter 3 project open, our first task will be to change the layout of Flash application so that there is room for the Description, Ingredients, and Instructions. We will start out by first increasing the size of our stage to 500 x 640. Once we have done this, we will need to resize our background so that it fits to the new stage. We will start this by first selecting the whole background region, and then converting that into a new **Movie Clip** by selecting **Modify | Convert to Symbol** from the Flash menu.



This will then bring up a new dialog, where we can give our new **Movie Clip** a name, which we will call **mcBackground**. We then need to make sure that we check the **Enable guides for 9-slice scaling**, which will allow us to resize the background without affecting the rounded edges.


Convert to Symbol

Name: mcBackground

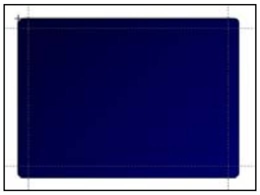
Type: Movie Clip

Registration:

Folder: Library root

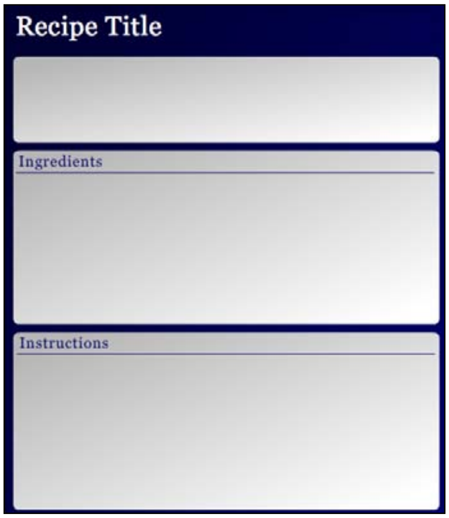
☑ Enable guides for 9-slice scaling

Once we create a new movie clip from our background, we will then enter this **Movie Clip** and then adjust the 9-scale guides so that they only cover the rounded edges.



We can now exit the background movie clip, and then resize the movie clip to a new height of 632 using the **Properties** panel.

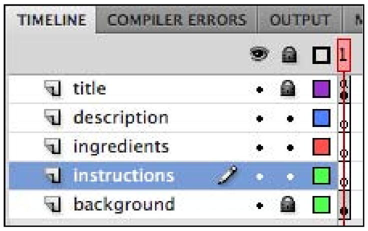
Our next task is to move the current title field to the top-left of our Flash application, and then create some background regions that will hold our new fields. The design of how this will look is completely up to you, but here is an illustration of what I just described:



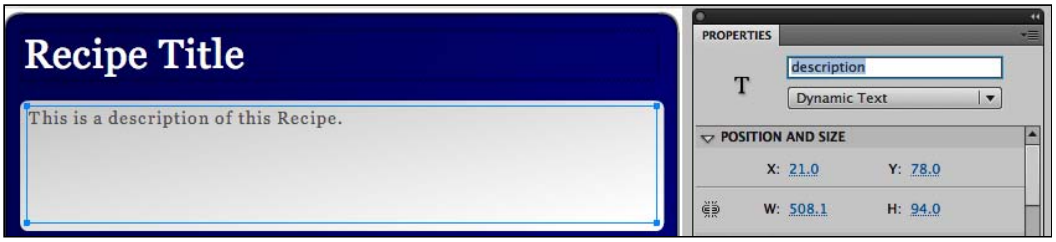
Now that our layout is ready for new content, the next step is to add new TextFields to hold our recipe content.

Adding dynamic TextFields for Drupal content

Since we have already added dynamic text fields in the previous chapters, we should be able to breeze through this section pretty quickly. The important thing to note here is that we will need to create a new layer for each text element within our Flash application, so that we can keep track of each field separately. We will do this within the timeline by creating three new layers for each of our new fields, and by then labelling them so that we can easily determine what they contain.



Now that we have each one of these separated, we can add new text fields in each layer, to be used for the **description**, **ingredients**, and the **instructions**. For each new Text field that we create, we will need to make sure to give them an instance name so that we can reference them within ActionScript. Each of these instance names should reflect the names of the fields that we created for our Recipe content type, which will be **description**, **ingredients**, and **instructions** respectively.

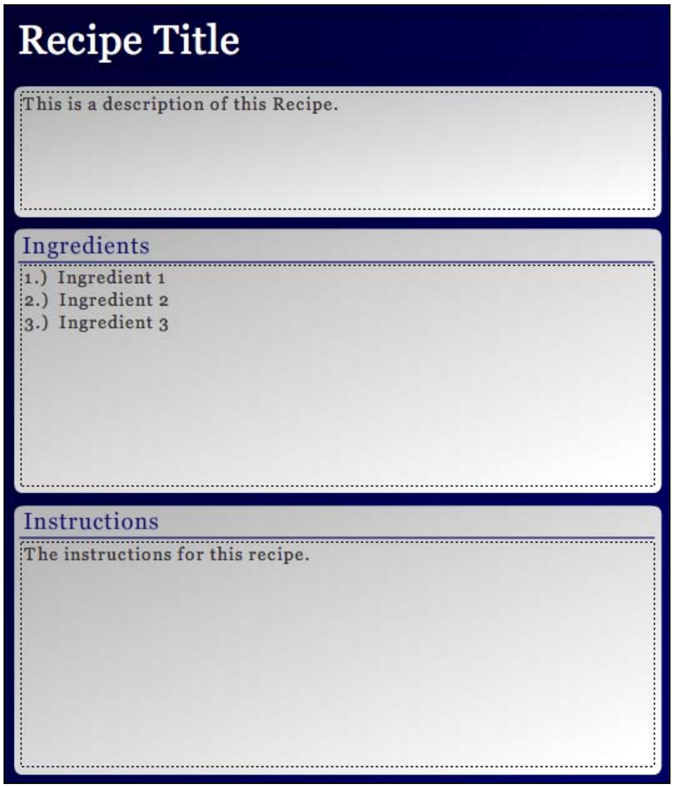


description

X: 21.0 Y: 78.0

W: 508.1 H: 94.0

When we are done, we should have something that resembles the following:



We are now ready to hook up these TextFields to real Drupal content.

Using ActionScript to show Drupal CCK fields

Since we are picking up where we left off from Chapter 2, we can use most of the ActionScript that was already written to show "Hello World". We can start this off by opening up our `main.as` file, and then we will shift our focus to the `onNodeLoad` function.

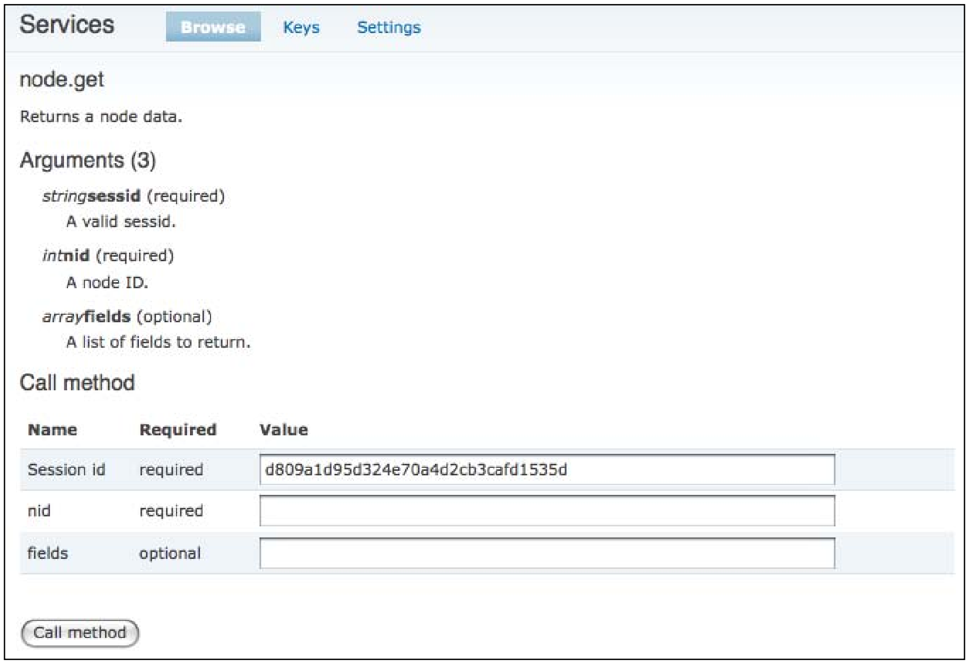
```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
}
```

This function gets called after our service call to Drupal's `node.get` service call and returns with the contents of the node. In the last chapter, we were simply using the node title to populate our `title` textbox that we had created within Flash. Now that we have new TextFields for each custom recipe field, we can use the node object, passed to the `onNodeLoad` function, to reference the data from these custom fields, and populate our TextFields with that data. Since the contents of this node object are somewhat a mystery, there is a fantastic tool that is provided with Drupal that will allow us to examine how this node is structured. We will then be able to use that information to fill out the contents of our `onNodeLoad` function to show our complete recipe node.

Using the Services Administrator

We now need to shift our focus back to Drupal, where we will navigate to the Service Administrator section by going to **Administer | Services**. The Services module comes equipped with a fantastic tool for analyzing any service routine when working with external applications. It allows for you to call any service routine, with any specified argument, and then see the result of that routine call. This can be used to easily analyze the data structure that our Flash application will receive after it makes a call to any of the service routines available.

Since we are using the **node.get** service routine to load each recipe node, we should be able to examine how the **Description**, **Ingredients**, and **Instructions** fields are represented, and then easily apply that to our Flash application. Let's do this by clicking on the link that says **node.get** in the node section. This will bring up the following page:



node.get

Returns a node data.

Arguments (3)

*string***sessid** (required)
A valid sessid.

*int***nid** (required)
A node ID.

*array***fields** (optional)
A list of fields to return.

Call method

Name	Required	Value
Session id	required	d809a1d95d324e70a4d2cb3cafd1535d
nid	required	
fields	optional	

The **Services** module automatically places a valid **Session id** in the session field, so we can just keep this field as it is. Because of this, all we really need to provide is the **nid** (node ID) of our Recipe node—since the **fields** field is optional.

> In order to determine the node ID for any node within the Drupal web site, simply navigate to **Administer | Content**, which will list all the content within the Drupal web site. The node ID can be found by hovering over any content link and then reading the last number in the URL. For example, if we hover over our Recipe node, we should see a URL similar to `http://localhost/drupal6/node/5`, which means that our node ID for this node is 5.

After we have the entered the node ID in the **nid** field, we can now click on the button that says **Call Method**. This will then show the results of that call within the **Results** section just below the **Call Method** button. To the untrained eye, this may look intimidating, but really what this is showing is the results for all the data contained within the recipe node that we just created, including the **Ingredients** and **Instructions**.

If we look within this data structure, we should see something that looks similar to the following:

```
[field_ingredients] => Array
(
   [0] => Array
   (
      [value] => 1 skinless, boneless chicken breast half
                 2 tablespoons minced green onion
                 2 tablespoons minced red bell pepper
                 3/4 cup shredded Monterey Jack cheese
                 5 (6 inch) flour tortillas

      [format] => 1
   )
)
```

Within our Flash application, we can now access the **Ingredients** field in the node object (which is what is returned when you call `node.get`). The ActionScript code to reference this field should look similar to the following:

```
node.field_ingredients[0]["value"]
```

Now, let's apply this concept to show the ingredients and instructions in our Flash application.

Showing CCK information in ActionScript

Let's move back to your Flash application and open up the `main.as` file. Since we now have an understanding of how the node data is structured, we can apply that knowledge to display the correct information within our Flash application. And, we will do this within the function `onNodeLoad`.

Showing the node description

Earlier in this chapter we changed our node type to represent the Body field as a description. Because of this change, it may be confusing at first when we access the Description field, simply because, the node object does not change the data structure to say description, but keeps it as body. Because of this, we will need to use the code `node.body` instead of `node.description` in order to access the description of our recipe node.

It is also very important to note that the Body/Description field for the node will be delivered as HTML text. If we set our description TextField with the contents of the node description, then we will end up showing the HTML tags along with the description contents. To solve this issue, Flash has added a new property to the TextField object that allows you to provide HTML text and it will parse the HTML and show that text accordingly. This property is called `htmlText`, and can be provided like the following:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;

   // Print out the description.
   description.htmlText = node.body;
}
```

Showing the ingredients and instructions CCK field

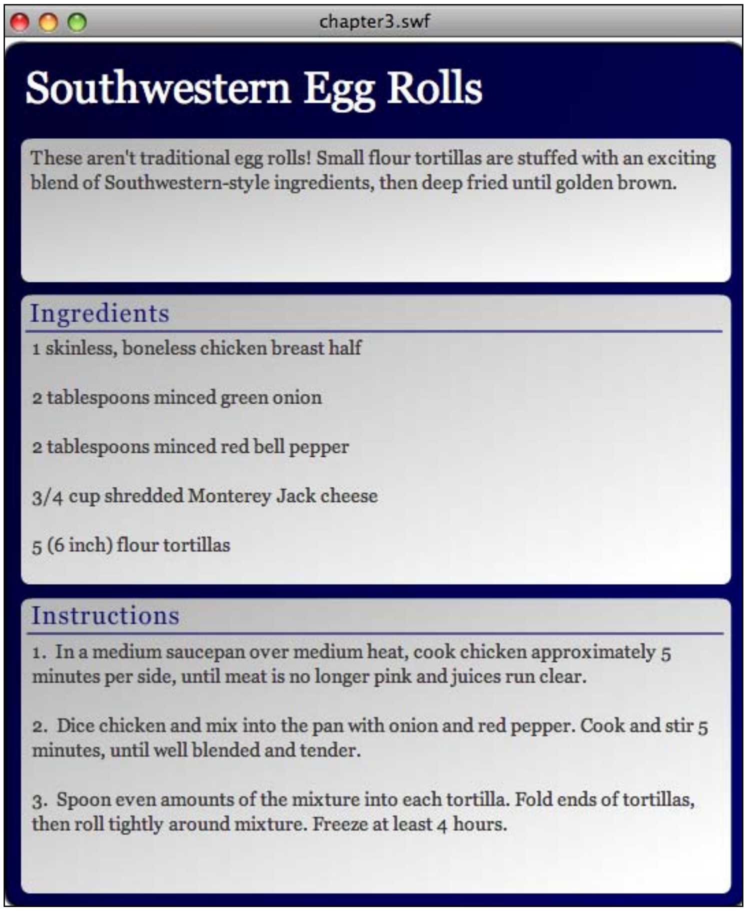
Since we have already examined the contents of the node object using the Drupal Services Administrator, we can populate our `ingredients` and `instructions` TextFields using the contents for each CCK field that we created. This will look as follows:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;

   // Print out the description.
   description.htmlText = node.body;

   // Show the ingredients and instructions.
   ingredients.text = node.field_ingredients[0]["value"];
   instructions.text = node.field_instructions[0]["value"];
}
```

Now that we have all the fields parsed correctly in ActionScript, we are ready to run our application to see it in action. But before we are able to test it, we will need to temporarily hard code the node ID variable at the top of the file, so that it reflects the recipe node ID that we just created in Drupal.

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";
var nodeId:Number = 5;
```

We can now finally test our Flash application, where we should then see something that looks like the following:



Southwestern Egg Rolls

These aren't traditional egg rolls! Small flour tortillas are stuffed with an exciting blend of Southwestern-style ingredients, then deep fried until golden brown.

Ingredients

1 skinless, boneless chicken breast half

2 tablespoons minced green onion

2 tablespoons minced red bell pepper

3/4 cup shredded Monterey Jack cheese

5 (6 inch) flour tortillas

Instructions

1. In a medium saucepan over medium heat, cook chicken approximately 5 minutes per side, until meat is no longer pink and juices run clear.

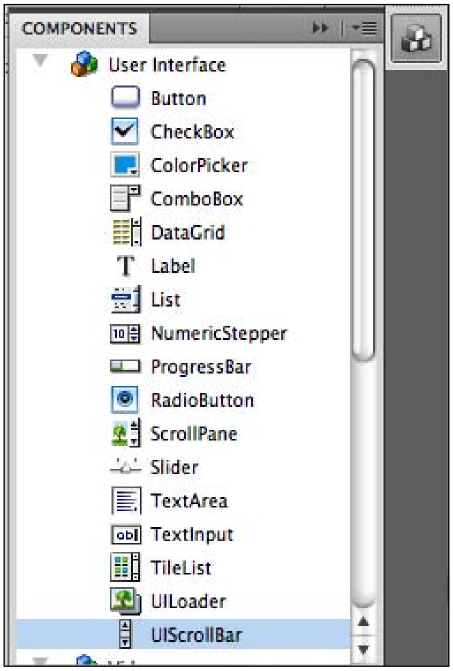
2. Dice chicken and mix into the pan with onion and red pepper. Cook and stir 5 minutes, until well blended and tender.

3. Spoon even amounts of the mixture into each tortilla. Fold ends of tortillas, then roll tightly around mixture. Freeze at least 4 hours.

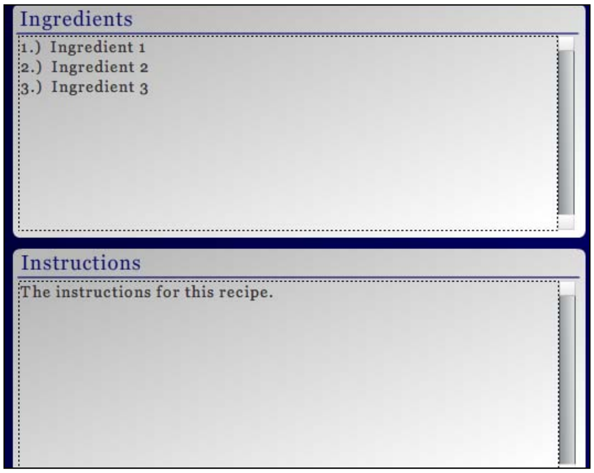
Although this is very cool, there is still a big question as to what we do when the ingredients and instructions text does not fit within the TextField region that we created. As it is right now, any text outside of this region will be clipped and therefore unreadable. Since we do not want this to happen, we can easily add some scroll bars to our text fields, so that it can hold any amount of text.

Adding ScrollBars to our TextFields

Surprisingly, thanks to the wonderful components that Flash provides out of the box, adding scroll bars to our text fields is not that difficult. To add a scroll bar to any text field, we will use the wonderful **UIScrollBar** component, which can be found in the components section of our Flash application. So, moving back to our `chapter3.fla` project, we will first open up the **COMPONENTS** section by clicking on the button in our window toolbar on the right-side of the Flash IDE.



We can then easily add a scroll bar to our TextFields by simply dragging a **UIScrollBar** component so that it overlaps with that TextField. Once we drop the scroll bar over the text region, it will snap into place telling us that a connection has been made. We will do this for both the **Ingredients** and **Instructions** TextFields as shown:



Ingredients

1.) Ingredient 1
2.) Ingredient 2
3.) Ingredient 3

Instructions

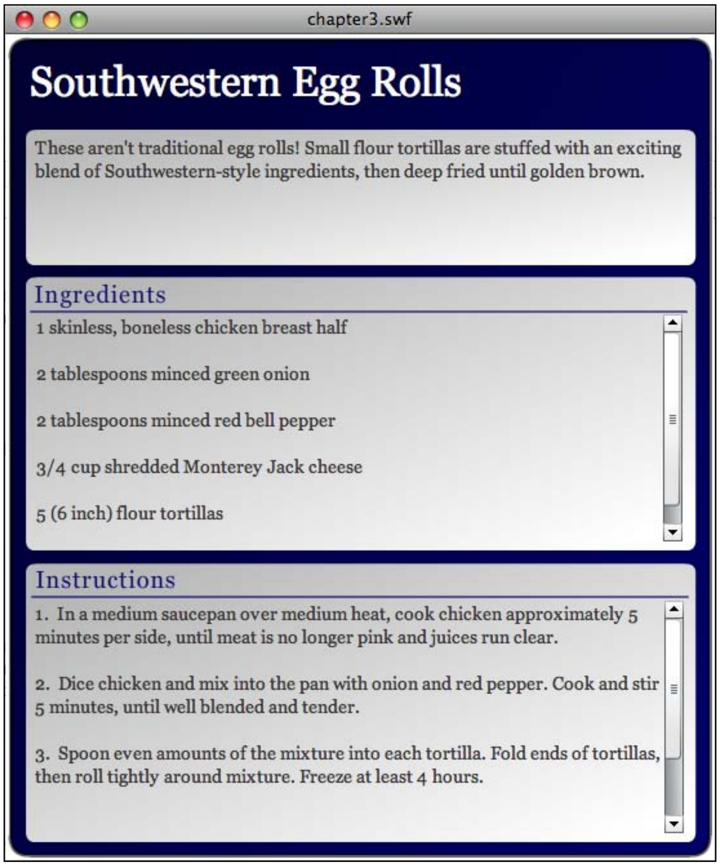
The instructions for this recipe.

After we have our scroll bars in place, the next step is to click on each one of them and give them an instance name using the **Properties** panel. We will call each one `ingredientScroll` and `instructionScroll` respectively.

Now that our scroll bars have instance names, we can add some simple code to our `onNodeLoad` function so that each scroll bar will refresh as the text is populated within them. We can do this by calling the `update` function on each `UIScrollBar` component as follows:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;

   // Print out the description.
   description.htmlText = node.body;

   // Show the ingredients and instructions.
   ingredients.text = node.field_ingredients[0]["value"];
   instructions.text = node.field_instructions[0]["value"];

   // Update the scroll bars.
   ingredientScroll.update();
   instructionScroll.update();
}
```

When we run our application again, we should be happy to see that our scroll bars show all the text for each CCK field within Drupal.



Southwestern Egg Rolls

These aren't traditional egg rolls! Small flour tortillas are stuffed with an exciting blend of Southwestern-style ingredients, then deep fried until golden brown.

Ingredients

1 skinless, boneless chicken breast half

2 tablespoons minced green onion

2 tablespoons minced red bell pepper

3/4 cup shredded Monterey Jack cheese

5 (6 inch) flour tortillas

Instructions

1. In a medium saucepan over medium heat, cook chicken approximately 5 minutes per side, until meat is no longer pink and juices run clear.

2. Dice chicken and mix into the pan with onion and red pepper. Cook and stir 5 minutes, until well blended and tender.

3. Spoon even amounts of the mixture into each tortilla. Fold ends of tortillas, then roll tightly around mixture. Freeze at least 4 hours.

Now that our Flash application works as we would expect, the next step is to change our `nodeId` variable back so that any node ID can be passed to our Flash application to show the recipe content of that node.

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";
var nodeId:Number = root.loaderInfo.parameters.node;
```

After we run the Flash application again, we are now ready to take our Recipe Flash application and use it within Drupal.

Creating a Drupal node template for Flash

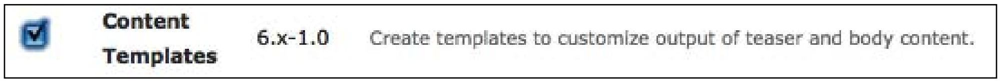
Moving back to Drupal, our next task is to take our compiled Recipe Flash application and integrate it into our Drupal system. We will start out by doing the same thing we did in the previous chapters, and add our Flash application to our Drupal web site using the **FlashNode** module.

We can do this by simply going to **Create Content | Flash**, and then selecting our new SWF file where it asks for the **Flash file**. We can then click on the **Save** button to submit this Flash application to our Drupal system. It is very important, at this point, that we remember the node ID that was created for our Flash node. We can determine the node ID by looking at the URL and taking note of the number that comes after `http://locahost/drupal6/node/`. With our Flash application in place, we now run into a unique situation that differs from the previous chapters.

In order to get our Recipe application to work as expected, we need to show the Flash application when anyone visits a recipe node. However, the `FlashNode` module attaches our SWF file to a completely separate node type called *Flash*, and not *Recipe*. In order to show that Flash application within a Recipe node type, we will need to create a node template for our Recipe node, where we will then reference the Flash application we just submitted using the `FlashNode` module. We can also utilize this template to pass in the node ID of the recipe that they are viewing. To accomplish this goal, we will use a very popular module for Drupal called Contemplate.

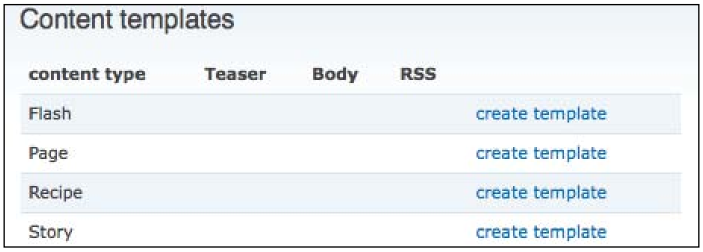
Using the Content Template module (Contemplate)

The Content Template module (Contemplate for short) is a wonderful module that will allow us to create a template for any content type in our system. A template is simply a way to tell the system how to display each content type when it is viewed by each visitor, and we can use this to tell Drupal to show our new Recipe Flash application when anyone views a Recipe node. To begin, we will first need to download and install this module by going to `http://www.drupal.org/project/contemplate`. Once we have placed this module in the `modules` folder, we can then install it by going to **Administer | Modules** and clicking on the **Content Templates** module within the **Content** section.



☑	Content Templates	6.x-1.0	Create templates to customize output of teaser and body content.

After we click on the **Save Configuration** button at the bottom of the **Modules** section, we can now navigate to **Administer | Content Templates**, where we will then be given an option to create a template for our **Recipe** node.



Content templates

content type	Teaser	Body	RSS	
Flash				create template
Page				create template
Recipe				create template
Story				create template

When we click on the **create template** link next to the Recipe node type, we should then be given a page that will let us customize how each recipe page will look. We can change the body of the Recipe nodes by expanding the **Body** section and then clicking on the checkbox that says **Affect body output**. At this point, we will get very creative in how we retrieve our Recipe Flash application, which we submitted earlier in this section. We can start by first loading the recipe Flash application node using the node ID that was created when we submitted our Recipe Flash application to Drupal. Assuming that our Flash node ID was 6, our code should then look like the following:

```
<?php
// Get the Flash application.
$sql = "SELECT * FROM {node} WHERE nid=%d";
$flash = db_fetch_object(db_query($sql, 6));
?>
```

Our next task is to add all the additional information to this Flash object by calling the `flashnode_load` function.

```
<?php
// Get the Flash application.
$sql = "SELECT * FROM {node} WHERE nid=%d";
$flash = db_fetch_object(db_query($sql, 6));
// Load the flash node.
flashnode_load( $flash );
?>
```

We can now add our FlashVars to this object by providing the node ID of the node that is being shown. The Contemplate module provides the `$node` object to each template so we have access to this node ID by referencing `$node->nid`.

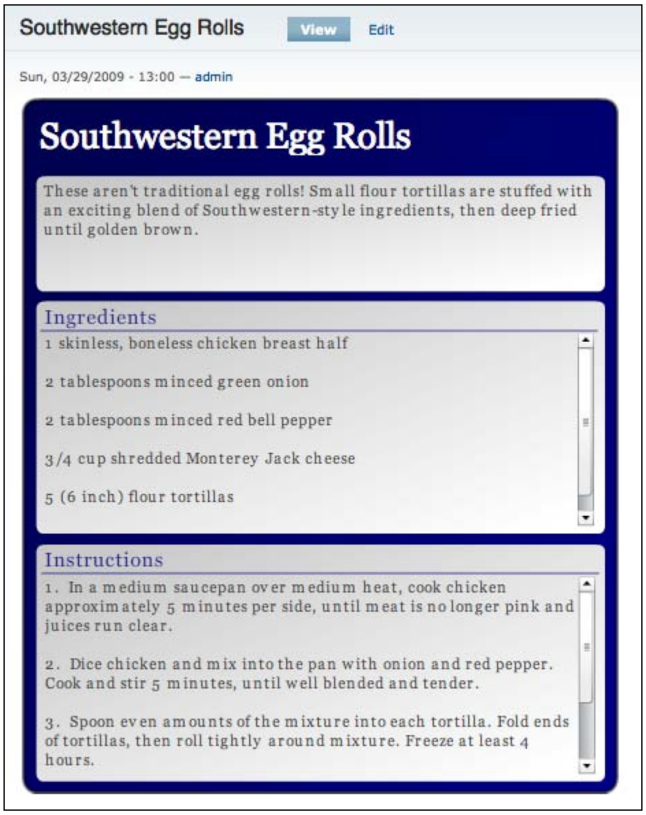
```
<?php
// Get the Flash application.
$sql = "SELECT * FROM {node} WHERE nid=%d";
$flash = db_fetch_object(db_query($sql, 6));
// Load the flash node.
flashnode_load( $flash );
```

```
// Set the flashvars to the right node Id.
$flash->flashnode["flashvars"] = 'node=' . $node->nid;
?>
```

Finally, we can show our Flash application using the `theme` function as follows:

```
<?php
// Get the Flash application.
$sql = "SELECT * FROM {node} WHERE nid=%d";
$flash = db_fetch_object(db_query($sql, 6));
// Load the flash node.
flashnode_load( $flash );
// Set the flashvars to the right node Id.
$flash->flashnode["flashvars"] = 'node=' . $node->nid;
// Show the Flash application.
print theme('flashnode', $flash->flashnode, FALSE);
?>
```

We can now save our template by clicking on the **Submit** button and then navigate to our Recipe node to see our Recipe Flash application in action!



Southwestern Egg Rolls

Sun, 03/29/2009 - 13:00 — admin

Southwestern Egg Rolls

These aren't traditional egg rolls! Small flour tortillas are stuffed with an exciting blend of Southwestern-style ingredients, then deep fried until golden brown.

Ingredients

1 skinless, boneless chicken breast half

2 tablespoons minced green onion

2 tablespoons minced red bell pepper

3/4 cup shredded Monterey Jack cheese

5 (6 inch) flour tortillas

Instructions

1. In a medium saucepan over medium heat, cook chicken approximately 5 minutes per side, until meat is no longer pink and juices run clear.

2. Dice chicken and mix into the pan with onion and red pepper. Cook and stir 5 minutes, until well blended and tender.

3. Spoon even amounts of the mixture into each tortilla. Fold ends of tortillas, then roll tightly around mixture. Freeze at least 4 hours.

Summary

In this chapter we learned how to incorporate custom data within our Flash applications by walking through a real Recipe web site example. By utilizing the power of CCK in our Flash applications, we have opened the door to a number of possibilities where Flash can be used to display dynamic content that is specific to any use case. In addition, we learned how to build a node template that allows us to use a Flash application as a direct replacement for the default HTML-driven content for each content type. In this chapter, we learned the following:

- Introduction to CCK and how to utilize its power within Flash
- How to add custom fields to each content type within our Drupal system
- How to change the default fields within Drupal
- Using the Services Administrator to extract node contents to be used in Flash
- Building a Flash application to use custom content
- Using a scroll bar to handle long text entries
- Adding a content template for our Flash applications

Although we covered much ground in this chapter with regard to text content, we still have not even scratched the surface for what power Flash can really deliver to our web site. In the following chapter, we will continue our journey by incorporating Drupal images into our Flash applications.

4

Drupal Images in Flash

Up to this point, we have learned how to build a Flash application using only text-based content from the Drupal CMS. Although text-based content is vital for any web application, it is the use of images that gets the attention of any person who is using your application, and that sets it apart from other applications. In this chapter we will learn how to build an image and content-rich Flash application by discussing the following topics:

- Image handling in Drupal
- Adding images to the Recipe content type
- Creating an image container in Flash
- Using ActionScript to load a Drupal image
- Resizing the image to fit inside our image container
- Preserving the Image ratio (scaling)
- Using ImageCache to dynamically resize images for Flash

Image handling in Drupal

I have heard from many Drupal beginners that image handling in Drupal can be a frustrating process. Not because Drupal lacks the ability to handle images, but simply because that person must pick between multiple contributed modules that all claim to do the same thing. This can be intimidating to a person who is exposed to Drupal for the first time. In this section, I will *not* go over all the possible ways to handle images and let you decide which one to use. Instead, I would like to present my opinion of a good image solution for Drupal, and then build our application from that approach. Please keep in mind that this approach is subjective and that there are other ways to do it, but for the sake of your sanity, I will stick with a single method for image handling, which is using the ImageField plug-in for CCK.

ImageField for CCK

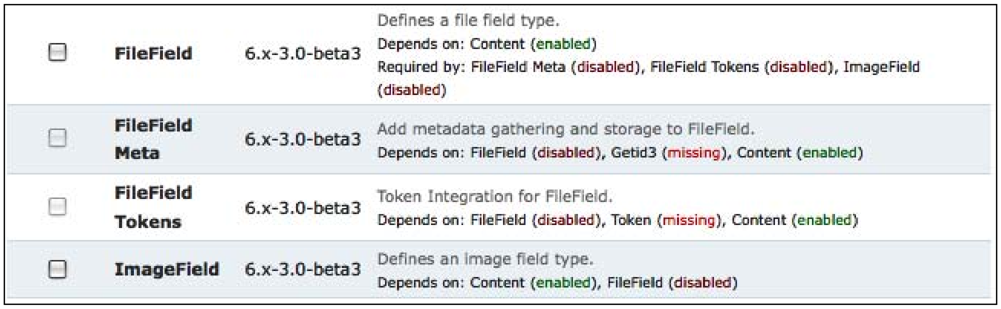
As far as I am concerned, **CCK** (**Content Construction Kit**) offers the most flexible method for adding custom content to our Drupal web site. Although there are dedicated modules that give us the ability to add custom content, I have found that it is easier and more consistent to take a single approach to create custom content through the use of CCK, and the ImageField module uses the power of CCK to add images to any content type in our system. I believe this flexibility is what gives ImageField an advantage over other popular image modules in Drupal. In this section, we will use the ImageField module to build onto the previous chapter by adding an image to the Recipe content type and then showing that image in our recipe Flash application.

Installing the ImageField module

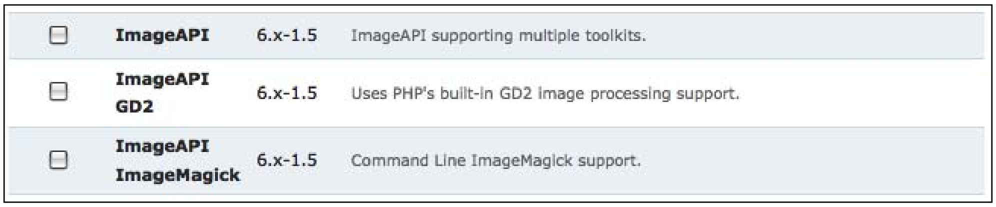
In order to successfully install the ImageField module, we will also need to install the modules that the ImageField module is dependent on. Because of this, we will first need to install a total of three modules in order to use the ImageField module. These modules are as follows:

- FileField—`http://www.drupal.org/project/filefield`
- ImageAPI—`http://www.drupal.org/project/imageapi`
- ImageField—`http://www.drupal.org/project/imagefield`

Once we place these three modules in our site's `modules` folder, we can navigate to **Administer | Modules**, where we should see the following:



☐	**FileField**	6.x-3.0-beta3	Defines a file field type. Depends on: Content (enabled) Required by: FileField Meta (disabled), FileField Tokens (disabled), ImageField (disabled)
☐	**FileField Meta**	6.x-3.0-beta3	Add metadata gathering and storage to FileField. Depends on: FileField (disabled), Getid3 (missing), Content (enabled)
☐	**FileField Tokens**	6.x-3.0-beta3	Token Integration for FileField. Depends on: FileField (disabled), Token (missing), Content (enabled)
☐	**ImageField**	6.x-3.0-beta3	Defines an image field type. Depends on: Content (enabled), FileField (disabled)



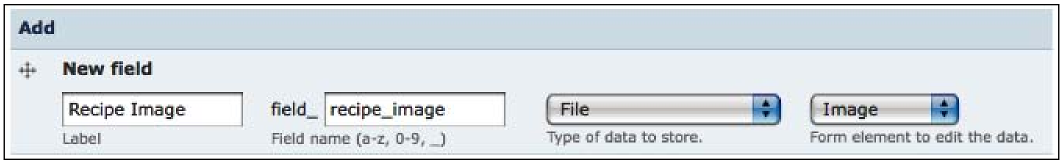
☐	**ImageAPI**	6.x-1.5	ImageAPI supporting multiple toolkits.
☐	**ImageAPI GD2**	6.x-1.5	Uses PHP's built-in GD2 image processing support.
☐	**ImageAPI ImageMagick**	6.x-1.5	Command Line ImageMagick support.

Given this list of modules, we will select the **FileField**, **ImageAPI**, **ImageAPI GD2**, and **ImageField** modules and then click on the **Save Configuration** button at the bottom of the page. Once we have done this, we will be able to add images to any content type in our Drupal system. Let's use this to add an image to our Recipe content type from the previous chapter.

Adding an Image field to our Recipe content type

Now that we have the ImageField module installed, we can navigate to the **Content Types** section in our Drupal administrator by going to **Administer | Content Types** and then take the following steps to set up a new **Image** field for our Recipe node type.

1. Click on the link that says **Manage Fields**, next to the Recipe content type.
2. In the **Add** section, let's provide the following information:



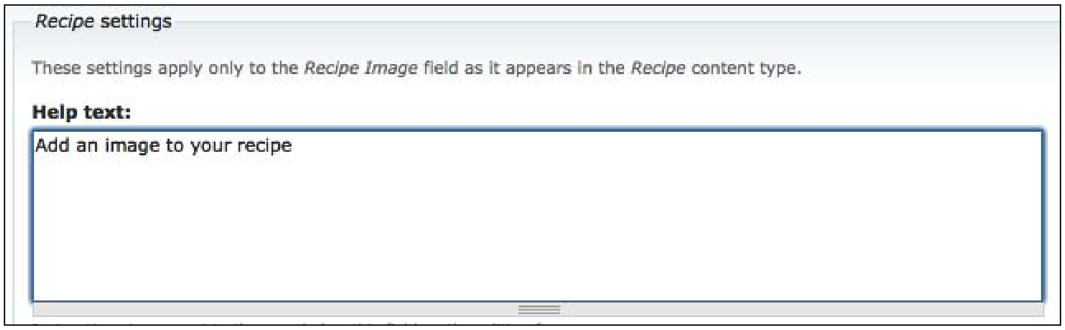
Add

New field

Recipe Image	field_ recipe_image	File	Image
Label	Field name (a-z, 0-9, _)	Type of data to store.	Form element to edit the data.

3. Now, click on the **Save** button to save our new field.

After we save this field, we will see a new page where we can configure our new image field. To set up our image field, we don't really need to provide anything here, except maybe the **Help text** as follows:



Recipe settings

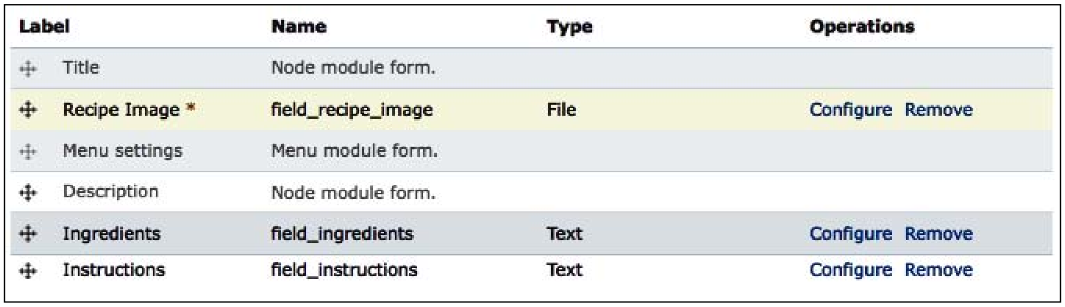
These settings apply only to the *Recipe Image* field as it appears in the *Recipe* content type.

Help text:

Add an image to your recipe

We can now save our new image field by clicking on the **Save field settings** button at the bottom of the page.

Now, the only thing left to do is move this field higher up in the field list. This will make it such that the image input from the user is not at the bottom of the form but closer to the top as you would see in any average recipe web site. We can do this by clicking on the ✥ symbol next to the recipe image field, and then dragging it below the **Title** field as follows:



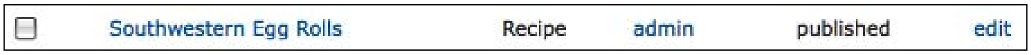
Label	Name	Type	Operations
✥ Title	Node module form.		
✥ Recipe Image *	field_recipe_image	File	Configure Remove
✥ Menu settings	Menu module form.		
✥ Description	Node module form.		
✥ Ingredients	field_ingredients	Text	Configure Remove
✥ Instructions	field_instructions	Text	Configure Remove

After we have moved our **Recipe Image** to where we want it in the node, we can commit that change by clicking on the **Save** button at the bottom of the page. Now, we can modify our recipe from Chapter 3 and add an image to our recipe.

Adding an image to our Recipe node

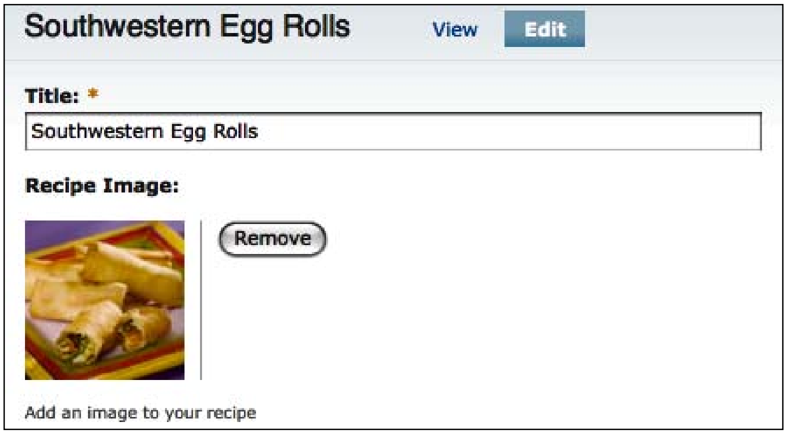
Since we will need to edit our recipe node from Chapter 2, we will need to first locate and edit our recipe content that we had previously submitted. The easiest way to do this is to examine all the content in our Drupal site and then select that recipe node in the content list. We can do this by going to **Administer | Content** in our Drupal administrator section.

This will then list all the content available in our site, where we should see the recipe we submitted from the previous chapter. We can edit this node by clicking on the **Edit** link, next to this listing:



□	Southwestern Egg Rolls	Recipe	admin	published	edit

Now that we are in the edit screen, we can add an image to our recipe by clicking the **Browse** button and selecting the image we would like to use for our recipe. Once the image is selected, we will need to attach this image to the node by clicking on the button that says **Upload**. After our image uploads, we should see something similar to the following:



Southwestern Egg Rolls

Title: *

Southwestern Egg Rolls

Recipe Image:

Add an image to your recipe

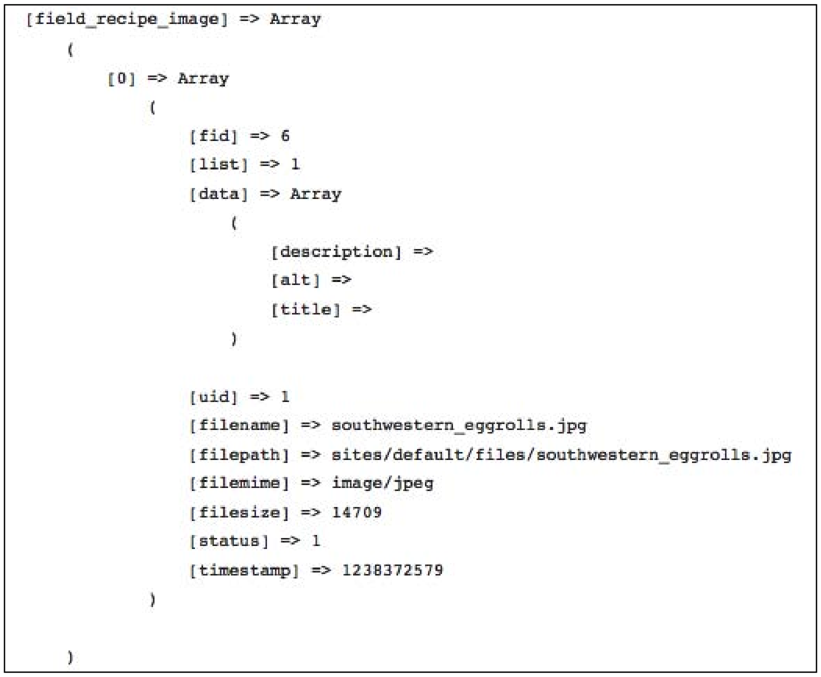
Once our image is attached to the node, we can now save this node by clicking on the button that says **Save** at the bottom of the page.

Don't be alarmed if you do not see your image on the screen after you save this node. Just because the image is now shown in the theme of the Recipe content type, does not necessarily mean that it is not attached to the node. We can verify if it is attached by using the Services Administrator.

Verifying that the image is attached

Before we navigate to the Services Administrator, we will again need to make a note of the node ID of the recipe that we attached our image to, which can be determined by looking at the URL for that page. Once we have this number in our head, we can then navigate to the Services Administrator by going to **Administer | Services**. As discussed in the previous chapter, this section will allow us to use any services that our system provides with the Services module. To verify that the image was attached to our node, we will click on the link that says **node.get**, which will then bring up the service page for the **node.get** routine. We can then place the node ID of the recipe that we just attached the image to, in the box that says **nid**, and then click on the **Call method** button to examine all the contents of the recipe node data structure.

Within the results of the method call, we should see the following:



```
[field_recipe_image] => Array
    (
        [0] => Array
            (
                [fid] => 6
                [list] => 1
                [data] => Array
                    (
                        [description] =>
                        [alt] =>
                        [title] =>
                    )

                [uid] => 1
                [filename] => southwestern_eggrolls.jpg
                [filepath] => sites/default/files/southwestern_eggrolls.jpg
                [filemime] => image/jpeg
                [filesize] => 14709
                [status] => 1
                [timestamp] => 1238372579
            )

    )
```

This shows that the recipe image field is present in our node, and that it can be accessed when the node object is provided to our `onNodeLoad` function. By looking at the information above, we can then determine our path to the image file by using the following code in ActionScript:

```
node.field_recipe_image[0]["filepath"];
```

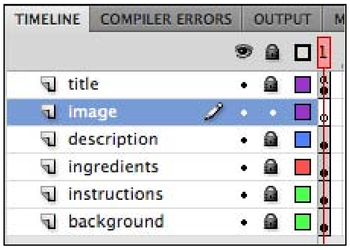
We can now use this code in our Flash application to show the image of our recipe.

Adding an image to our Recipe Flash application

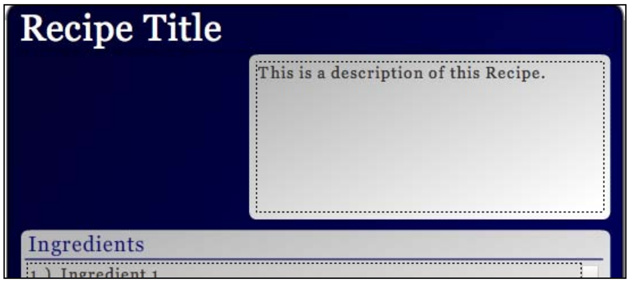
Our next step is to show this image in the Recipe Flash application that we built in the previous chapter. But before we begin, we will need to copy the `chapter3` directory and its contents and then paste that directory as `chapter4`. Once we have done this, we will then need to rename the `chapter3.fla` file within this new directory to `chapter4.fla`, and then open that project up in Flash. We will now expand our Recipe application to include an image from Drupal. Once we have this project open, we will start by adding a MovieClip container where our image will be shown.

Adding a MovieClip container for our image

The first step when adding any new element to a Flash application is to create a new layer in the timeline to place the objects that will be used for that element. So, shifting our focus up to the timeline, let's create a layer called **image**, and place it just above the **description** layer as shown:



Before we can add the image to our stage, we must first make sure we provide room for the image to be shown. Looking at the layout of our Recipe application, a good place for an image will most likely be to the left of the description. So, let's unlock the description layer and change the TextField and background sizes to make room for an image as shown:



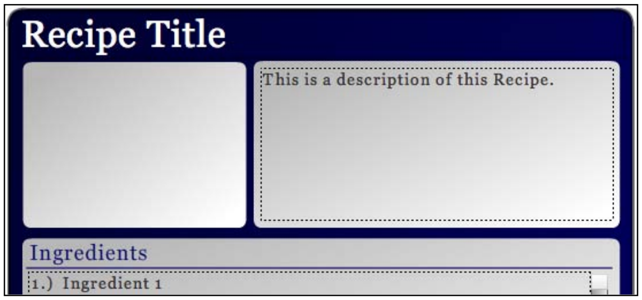
Recipe Title

This is a description of this Recipe.

Ingredients

1.) Ingredient 1

Once we are done with making room for our image, we can now lock all the layers except for the image layer, and then add rectangle object where we would like to show the image using the □ from the toolbar.



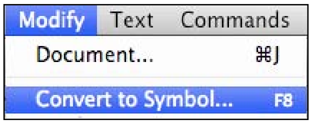
Recipe Title

This is a description of this Recipe.

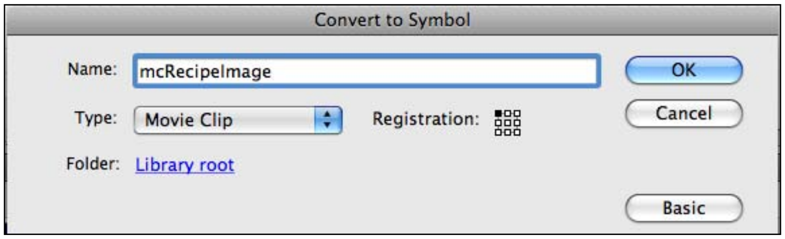
Ingredients

1.) Ingredient 1

In order to reference this area within our ActionScript, we need to convert this rectangle into a MovieClip. This can be done by selecting the rectangle object you just created (using the ↖ from the toolbar) and clicking on the rectangle we just created. Once it is selected, you can then go to the **Modify** menu item and select **Convert to Symbol**.



This will bring up another window, where we can give our **Movie Clip** a name as follows:



Name: mcRecipeImage
Type: Movie Clip
Registration:
Folder: Library root

After you have finished converting our rectangle object into a **Movie Clip**, we can then give it an instance name using the **PROPERTIES** section. Let's call our instance, **image**.

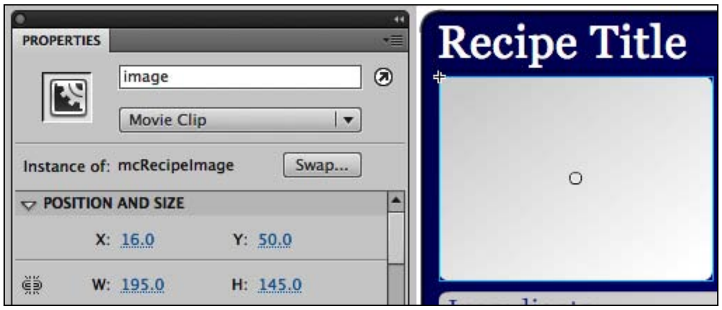


image
Movie Clip
Instance of: mcRecipeImage
X: 16.0 Y: 50.0
W: 195.0 H: 145.0

Now that our image MovieClip has been added, our next task is to use ActionScript to load the image from Drupal's ImageField into this movie clip object.

Using ActionScript to load the Recipe image

In order to load our image into the movie clip that we just created, we will need to tap into the ActionScript code where the node has finished loading and passes the node information to the `onNodeLoad` function. As it stands right now, this function looks like the following within our `main.as` file:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;

   // Print out the description.
   description.htmlText = node.body;

   // Show the ingredients and instructions.
   ingredients.text = node.field_ingredients[0]["value"];
   instructions.text = node.field_instructions[0]["value"];

   // Update the scroll bars.
   ingredientScroll.update();
   instructionScroll.update();
}
```

We are now ready to add our code to this function to load an image provided from Drupal. To do this, we will start with setting up the path to our image.

Working with the Image path

Since we have already validated that our image was attached to the recipe using the Services Administrator, we now have access to this image path by referencing our image field in the node object as follows:

```
node.field_recipe_image[0]["filepath"];
```

The only problem with this code is that the value of the `filepath` provided from the image field is relative to the base installation of our Drupal web site. This means that our image path may look like `sites/default/files/image.jpg`. In order for Flash to consistently load an image from a remote source, it must first have an absolute path to the image it wishes to load. Converting the relative path given from the image field to an absolute path would then look something like `http://www.mysite.com/sites/default/files/image.jpg`. In order to perform this conversion from relative to absolute paths, we will need to modify the code above so that we can provide the URL of our web site and then append the relative path to that URL.

Given that we have already defined the baseURL variable at the top of the main.as file, we can fix our path to include the absolute path using the following code within the onNodeLoad function.

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
    // Print out the description.
   description.htmlText = node.body;
    // Show the ingredients and instructions.
   ingredients.text = node.field_ingredients[0]["value"];
   instructions.text = node.field_instructions[0]["value"];
    // Update the scroll bars.
    ingredientScroll.update();
    instructionScroll.update();
    // Load the image
    var imagePath:String = baseURL;
    imagePath += "/";
    imagePath += node.field_recipe_image[0]["filepath"];
   trace( imagePath );
}
```

At this point, we are doing nothing with this image path besides just displaying it within our debugger using the trace function. Our next step here is to display the image by replacing the trace function call with a new function that we will create called loadImage, which will take the path of the image as an argument and then load that image into our MovieClip container.

Creating a loadImage function

It is considered good code practice to break apart different elements of functionality into their own functions. This makes the code more readable, reusable, and maintainable. Loading an image is something that would definitely fit this criterion and, therefore, should get its own function, which we will then call within our onNodeLoad function. Our first task will be to create what is called a stub function, which will simply be a placeholder to the ActionScript that will load the image into our MovieClip. We can do this by simply replacing the trace statement, in the code from the previous section, with our own custom function called loadImage, and then defining a simple function below the onNodeLoad routine as follows:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   ...
   ...
```

```
      // Load the image
      var imagePath:String = baseURL;
      imagePath += "/";
      imagePath += node.field_recipe_image[0]["filepath"];
      loadImage( imagePath );
   }
   // Load an image into a movie clip.
   function loadImage( filepath:String )
   {
   }
```

We are now ready to take this one step further and load the image of the path given to this function into our image MovieClip.

Loading an Image

In order to load an image in ActionScript, we will need to utilize a standard ActionScript 3 class called `Loader`. The `Loader` class is used to load files into our Flash application including images. In order to use this class, we will first need to declare an instance, which we will just call `imageLoader`. Since this variable will eventually be needed within multiple places in our `main.as` file, it will need to be a global variable, which means it will need to be accessible within the entire ActionScript code. To create a global variable, we will need to place it at the top of the `main.as` file, outside of any function. A good place for us to declare this variable is below the `nodeId` variable that we have already defined.

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";
var nodeId:Number = root.loaderInfo.parameters.node;

// Declare our imageLoader
var imageLoader:Loader;
```

Now that we have the `imageLoader` declared, we can place the instantiation of this variable within our `loadImage` function. We can also add a check to make sure our image path is valid before instantiating this loader variable.

```
// Load an image into a movie clip.
function loadImage( filepath:String )
{
   // If the filepath exists...
   if( filepath ) {
         // Instantiate our loader.
         imageLoader = new Loader();
   }
}
```

Now that we have this loader declared and instantiated, our next step is to add the Event listeners, which will be used to handle the events when the image has finished loading, and in the event that an error has occurred. To do this, we will need to create a new function called `onImageLoaded`, which we can place directly below the `loadImage` function. This new function will be a callback function that will be called when the image has finished loading. As for handling error conditions, we have already defined a `callback` function in the previous section called `onError` that we can use for this image loader also.

```
// Load an image into a movie clip.
function loadImage( filepath:String )
{
   // If the filepath exists...
   if( filepath ) {
      // Instantiate our loader.
      imageLoader = new Loader();
      // Add our event listeners.
      imageLoader.contentLoaderInfo.addEventListener( Event.COMPLETE,
      onImageLoaded );
      imageLoader.contentLoaderInfo.addEventListener
      (IOErrorEvent.IO_ERROR, onError);
      imageLoader.addEventListener(IOErrorEvent.IO_ERROR, onError);
   }
}
// Called when an image has finished loading.
function onImageLoaded( event:Event )
{
}
```

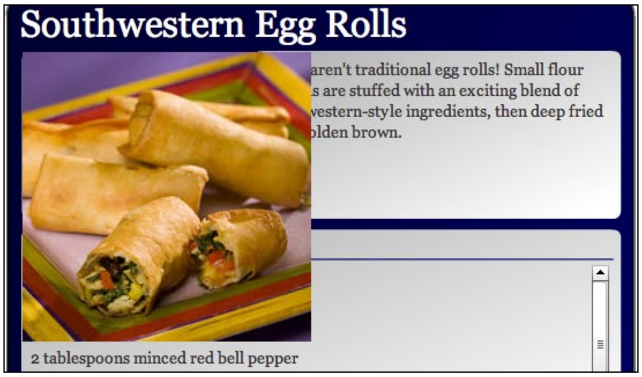
The last and final step to load an image is to add this loader to the MovieClip that we created, which will hold the image followed by a simple call to load the image. To add the loader to the MovieClip, we can use the standard Flash function called `addChild`, which is used to add any Object to a MovieClip. As for loading the image, we will use the `load` function on the `imageLoader` object, using another standard class in Flash called `URLRequest`. The `URLRequest` class is used to handle all communication between two remote locations, as well as enforce cross-domain policies that Flash has introduced for security reasons. To use this class, we will just pass the path of our image into the constructor of the `URLRequest` and then pass that object to the `load` function of our `imageLoader`. Below, you will find the complete `loadImage` function and `onImageLoaded` callback function:

```
// Load an image into a movie clip.
function loadImage( filepath:String )
{
   // If the filepath exists...
   if( filepath ) {
      // Instantiate our loader.
      imageLoader = new Loader();
      // Add our event listeners.
      imageLoader.contentLoaderInfo.addEventListener
      ( Event.COMPLETE, onImageLoaded );
       imageLoader.contentLoaderInfo.addEventListener
       (IOErrorEvent.IO_ERROR, onError);
      imageLoader.addEventListener(IOErrorEvent.IO_ERROR, onError);
       // Add this loader to the image MovieClip.
       image.addChild( imageLoader );
       // Load the image.
       imageLoader.load(new URLRequest(filepath));
    }
}
// Called when an image has finished loading.
function onImageLoaded( event:Event )
{
}
```

We are now ready to run this application to test its functionality, but we will first need to hard code our nodeId variable back to the number value of our Recipe node that we created in Drupal.

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";
var nodeId:Number = 5;
```

We can now run this application and see our Drupal image load into our MovieClip. However, once the application loads, you will most likely notice an obvious flaw in that our image does not fit within the MovieClip container that we created for it.



Southwestern Egg Rolls

aren't traditional egg rolls! Small flour
s are stuffed with an exciting blend of
vestern-style ingredients, then deep fried
olden brown.

2 tablespoons minced red bell pepper

Obviously, we still have some work to do here to resize this image to the correct dimensions.

Resizing an image

One thing to note about our method for loading an image is that once the image has been loaded into the image MovieClip, it will automatically resize the MovieClip to hold that image. The approach we need to take here is to resize the MovieClip back to its original size after the image has loaded. This will require us to keep a global variable that stores the size of the image MovieClip before we even start to load it. We can do this by declaring a rectangle variable that we will set within the `loadImage` function, and then use again to resize the image in the `onImageLoaded` function. Since this variable is needed within two different functions, it will also need to be a global variable, which we can place below the `imageLoader` variable that we have already defined.

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";
var nodeId:Number = 5;

// Declare our imageLoader and imageSize
var imageLoader:Loader;
var imageSize:Rectangle;
```

Since this variable was declared as global, we can now use it within any function and they will all reference the same variable. We can start this by storing the original size of the image within the `loadImage` function:

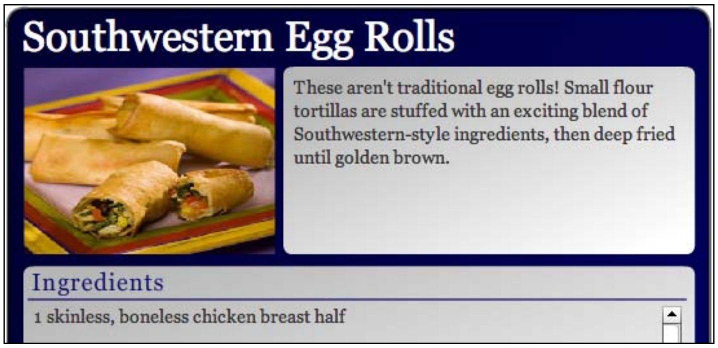
```
// Load an image into a movie clip.
function loadImage( filepath:String )
{
   // If the filepath exists...
      if( filepath ) {
       // Instantiate our loader.
      imageLoader = new Loader();
      // Add our event listeners.
      imageLoader.contentLoaderInfo.addEventListener
      ( Event.COMPLETE, onImageLoaded );
      imageLoader.contentLoaderInfo.addEventListener
      (IOErrorEvent.IO_ERROR, onError);
      imageLoader.addEventListener(IOErrorEvent.IO_ERROR, onError);
      // Store the size of the image before loading.
         imageSize = new Rectangle
         ( image.x, image.y, image.width, image.height );
     // Add this loader to the image MovieClip.
```

```
      image.addChild( imageLoader );
      // Load the image.
      imageLoader.load(new URLRequest(filepath));
   }
}
```

Now that we have stored the size of our MovieClip before we load the image, we can now resize the MovieClip back to its original size after the image loads. We can do this within the `onImageLoaded` function.

```
// Called when an image has finished loading.
function onImageLoaded( event:Event )
{
   // Resize our image back to its original size.
   imageLoader.width = imageSize.width;
   imageLoader.height = imageSize.height;
}
```

When we run our application now, you will see that it looks much better than before!



Southwestern Egg Rolls

These aren't traditional egg rolls! Small flour tortillas are stuffed with an exciting blend of Southwestern-style ingredients, then deep fried until golden brown.

Ingredients

1 skinless, boneless chicken breast half

Although this may look good, under close inspection we can see that there is a slight horizontal distortion in our image. The reason for this distortion is because we are not preserving the ratio (width/height) of the image when it is resized. From looking at the code above, we are assuming that the ratio of the MovieClip that we created to hold the image will have the exact same ratio as the image we are loading. To solve this problem, we will need to revise our code above to account for the ratio of the loaded image and use that to properly resize our MovieClip according to the loaded image's width and height ratio.

Preserving the width and height ratio (scaling)

In order to preserve the image ratio when loading our recipe image, we will need to apply the same width/height ratio to our MovieClip after the image has loaded. Our first task will be to determine the image ratio of the loaded image. This can be accomplished by examining the event object that was passed to the `onImageLoaded` function by the event handler. This event object contains a property that points to the object that triggered the event, which is called the `target`. This just so happens to be a pointer to the image that was loaded in our MovieClip, and thus can be used to determine the ratio by dividing the `width` by the `height` of the `target`. We will first delete our previous code within the `onImageLoaded` function, and then replace it with this functionality as follows:

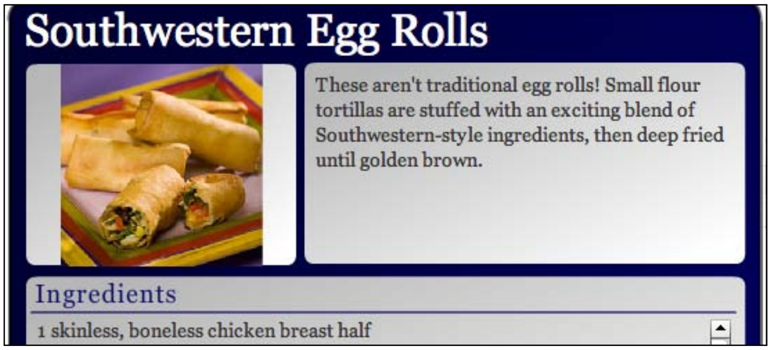
```
// Called when an image has finished loading.
function onImageLoaded( event:Event )
{
   // Determine our image ratio.
   var imageRatio:Number = event.target.width / event.target.height;
}
```

Our next task is to use this ratio and apply that when resizing the MovieClip container back to its original size. We can do this by calculating the ratio of our MovieClip and then building a scaled rectangle that will represent the dimensions of the image that will fit within our MovieClip. Our goal here is to pick the largest image size possible, where its width and height can both fit within the confines of the original MovieClip size. The code to do this might be intimidating, so I have tried my best to comment the code to illustrate what each line is doing.

```
// Called when an image has finished loading.
function onImageLoaded( event:Event )
{
   // Determine our image ratio.
   var imageRatio:Number = event.target.width / event.target.height;
   // Set up our scaled rectangle by initializing it
   // to the MovieClip size.
   var scaledRect:Rectangle = new Rectangle
   ( imageSize.x, imageSize.y, imageSize.width, imageSize.height );
   // Determine our MovieClip ratio.
   var mcRatio:Number = (imageSize.width / imageSize.height);
   // If the MovieClip ratio is greater than the image Ratio.
   if( mcRatio > imageRatio )
   {
      // Set the scaled rect to be the same as the MovieClip height.
      scaledRect.height = imageSize.height;
```

```
         // The width is the MovieClip height multiplied
         // by the image ratio.
         scaledRect.width = Math.floor(imageSize.height * imageRatio);
      }
      else
      {
         // The scaled rectangle is the MovieClip width divided
         // by the image ratio.
         scaledRect.height = Math.floor(imageSize.width / imageRatio);

         // The scaled rectangle is the same as the MovieClip width.
         scaledRect.width = imageSize.width;
      }

      // Resize and center our image.
      imageLoader.x += (imageSize.width - scaledRect.width) / 2;
      imageLoader.y += (imageSize.height - scaledRect.height) / 2;
      imageLoader.width = scaledRect.width;
      imageLoader.height = scaledRect.height;
   }
```

Once we run our Flash application, we will see that the image has been resized according to the correct image ratio. This will allow Drupal to load any size image into your Flash application and then automatically resize those images, while at the same time, maintain the original aspect ratio.



Southwestern Egg Rolls

These aren't traditional egg rolls! Small flour tortillas are stuffed with an exciting blend of Southwestern-style ingredients, then deep fried until golden brown.

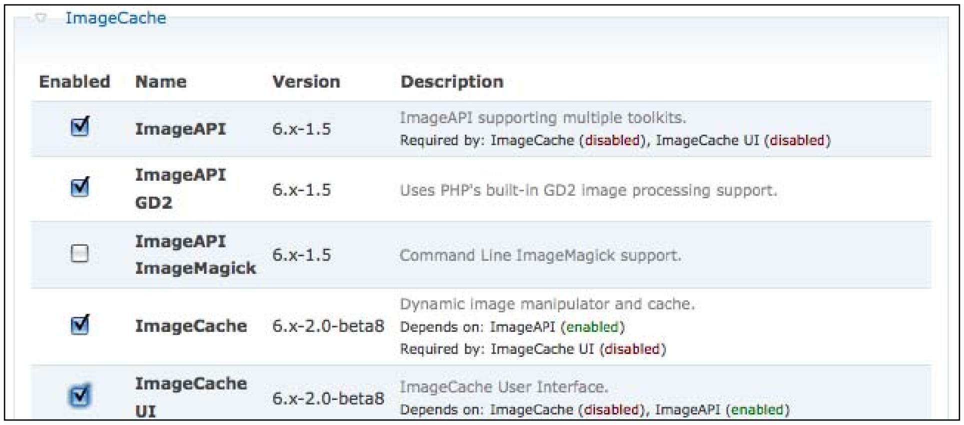
Ingredients

1 skinless, boneless chicken breast half

Using this method allows for our Flash application to handle larger images from Drupal, but you can probably imagine how this is very inefficient since loading large images can slow down any application. We can take this one step further by dynamically resizing our images from within Drupal before they are even loaded into our Flash application. This will clearly illustrate the power we get when combining Flash with a Content Management System. The module that we will use to achieve this dynamic image resizing is the fantastic **ImageCache** module.

Using Drupal's ImageCache with Flash

Before we begin with this section, we will first need to go to `http://www.drupal.org/project/imagecache` and download and install the **ImageCache** module into our Drupal installation.

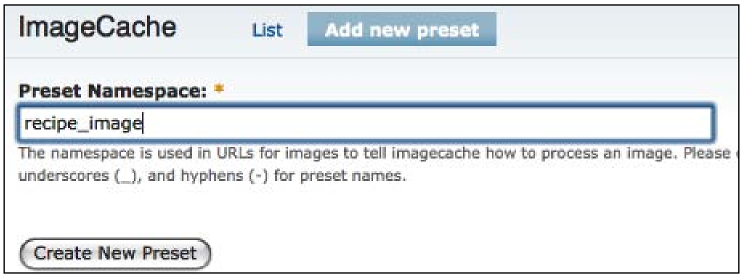


Enabled	Name	Version	Description
☑	ImageAPI	6.x-1.5	ImageAPI supporting multiple toolkits. Required by: ImageCache (disabled), ImageCache UI (disabled)
☑	ImageAPI GD2	6.x-1.5	Uses PHP's built-in GD2 image processing support.
□	ImageAPI ImageMagick	6.x-1.5	Command Line ImageMagick support.
☑	ImageCache	6.x-2.0-beta8	Dynamic image manipulator and cache. Depends on: ImageAPI (enabled) Required by: ImageCache UI (disabled)
☑	ImageCache UI	6.x-2.0-beta8	ImageCache User Interface. Depends on: ImageCache (disabled), ImageAPI (enabled)

The **ImageCache** module is basically a dynamic image manipulation module that we will use with our Flash application to ensure that our recipe image is not too large when we load it into our application. This is critical in making sure that bandwidth is preserved and our Flash application is not loading any unnecessarily large images, which will also improve the speed at which our images load into our Recipe node. After we have this module installed by clicking on the **Save Configuration** button, we will then need to navigate to the **ImageCache** administrator section found at **Administer | ImageCache**, where we will create our recipe image preset.

Creating an ImageCache preset

Once we are in the **ImageCache** administrator section, we can create a preset for our Recipe content type by clicking on the link that says **Add new preset**, which will then take us to a new page where we can enter our preset name. Let's call our recipe image preset **recipe_image**.



Preset Namespace: *

recipe_image

The namespace is used in URLs for images to tell imagecache how to process an image. Please
underscores (_), and hyphens (-) for preset names.

After we accept this name by clicking on the **Create New Preset** button, we will see a new page where we can select different actions to perform on our recipe image. Since we will want to scale our image to a certain width and height, while maintaining aspect ratio, we can select the **Add Scale** action from the **New Actions** listing.

Within the width and height input boxes, we can either provide a percentage scale (using the %), or we can provide a maximum dimension in pixels to scale the image. Since we already know the maximum size of our image from our MovieClip, we can provide those exact values into the **ImageCache** width and height settings on this page. So, going back to our Flash application, we can determine the width and height of our MovieClip by looking at the **Properties** section. We will now want to copy over those values into the width and height boxes for our **ImageCache** settings as shown:

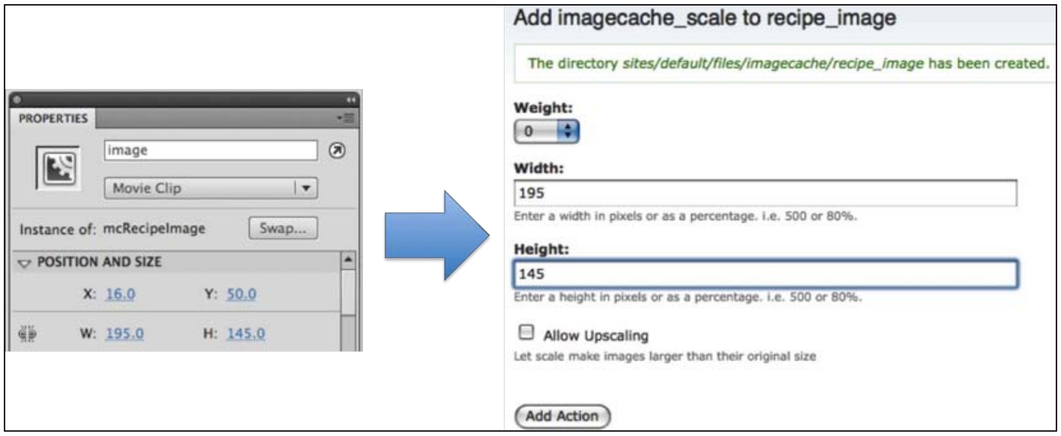


image
Movie Clip
Instance of: mcRecipeImage
X: 16.0 Y: 50.0
W: 195.0 H: 145.0

Add imagecache_scale to recipe_image

The directory *sites/default/files/imagecache/recipe_image* has been created.

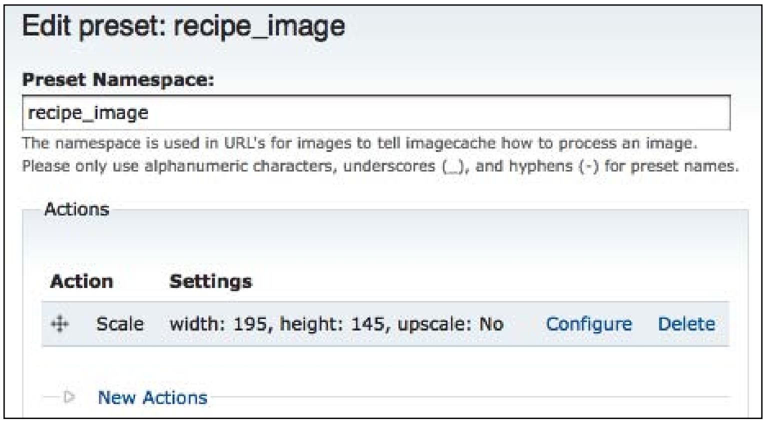
Weight: 0

Width: 195
Enter a width in pixels or as a percentage. i.e. 500 or 80%.

Height: 145
Enter a height in pixels or as a percentage. i.e. 500 or 80%.

□ Allow Upscaling
Let scale make images larger than their original size

Now that we have provided the correct dimensions for our ImageCache scaling, we will save this by clicking on the **Add Action** button on the ImageCache scaling page, which will then show your field added to the **Actions** panel.



Edit preset: recipe_image

Preset Namespace:
recipe_image
The namespace is used in URL's for images to tell imagecache how to process an image.
Please only use alphanumeric characters, underscores (_), and hyphens (-) for preset names.

Actions

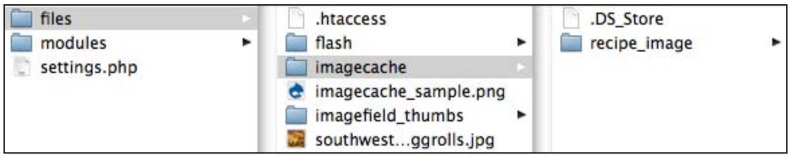
Action	Settings
Scale	width: 195, height: 145, upscale: No

We now need to make sure that this change commits by clicking on the **Update Preset** button. At this point, our configuration for the recipe image scaling is complete. We can now shift our focus back to Flash, where we will modify our code to load the ImageCache image instead of the original image provided from the ImageField module.

Adding an ImageCache image in Flash

Adding an ImageCache image in Flash might be a little trickier than expected, simply because the manipulated image path is not provided to us within the node object like the ImageField image. Because of this, we will need to understand how ImageCache stores all the converted images, and then modify our Flash application to use the ImageCache version.

ImageCache uses directory hierarchy to categorize all dynamically manipulated images. Fortunately, this can be used in our Flash application to point to the right image. In the previous section, we created a preset called **recipe_image**, which we used to add a resize action to all images submitted for the Recipe content type. The ImageCache module uses this name as a sub directory within an **imagecache** directory in the Drupal files directory as shown in the following directory tree:



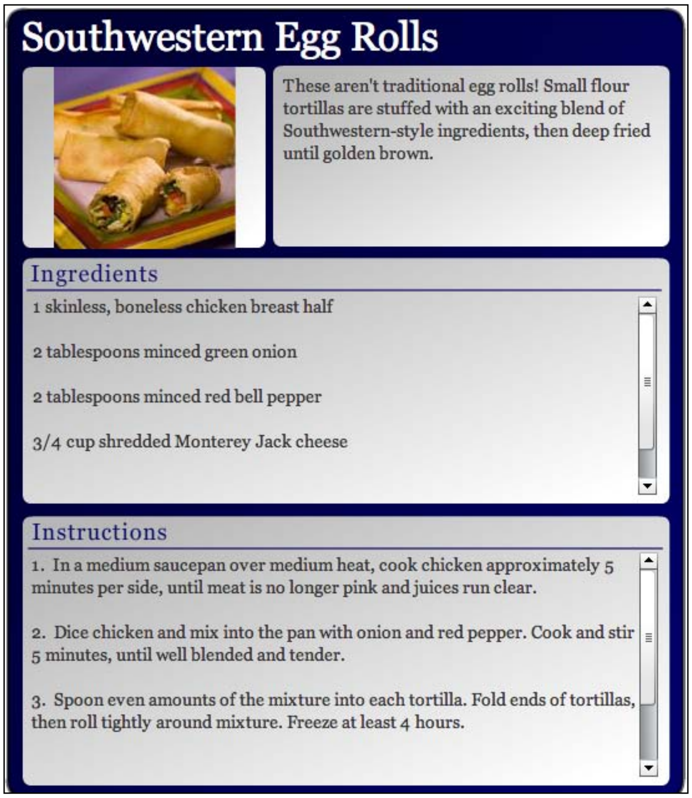
Within this folder, it will then place the dynamically converted image using the same file name as the original. This is a good thing, because we can now create a path within ActionScript that can link to the ImageCache image, since we can determine the filename from our node object.

Changing our ActionScript for ImageCache

Moving back to our `main.as` file, we can now construct the correct path needed to reference the `ImageCache` version of our images by modifying our previous `imagePath` variable. Instead of referencing the `filepath` for our image, we can now include the new `ImageCache` base path and then append the `filename` onto that path. This will then create the path we need to reference the dynamically resized image from `ImageCache`.

```
// Load the image
var imagePath:String = baseURL;
imagePath += "/sites/default/files/imagecache/recipe_image/";
imagePath += node.field_recipe_image[0]["filename"];
loadImage( imagePath );
```

At this point, we can run our Flash application and should then see the ImageCache image load instead of the full size image.



Southwestern Egg Rolls

These aren't traditional egg rolls! Small flour tortillas are stuffed with an exciting blend of Southwestern-style ingredients, then deep fried until golden brown.

Ingredients

1 skinless, boneless chicken breast half

2 tablespoons minced green onion

2 tablespoons minced red bell pepper

3/4 cup shredded Monterey Jack cheese

Instructions

1. In a medium saucepan over medium heat, cook chicken approximately 5 minutes per side, until meat is no longer pink and juices run clear.

2. Dice chicken and mix into the pan with onion and red pepper. Cook and stir 5 minutes, until well blended and tender.

3. Spoon even amounts of the mixture into each tortilla. Fold ends of tortillas, then roll tightly around mixture. Freeze at least 4 hours.

Although this may look like there is no difference using ImageCache and not using ImageCache; the huge difference here is efficiency. By dynamically resizing our image to fit the size of our Flash image area, we are not taking on any bandwidth overhead of loading an image larger than the region needs to show. This can then be easily translated into money saved since that bandwidth can be used to service other clients.

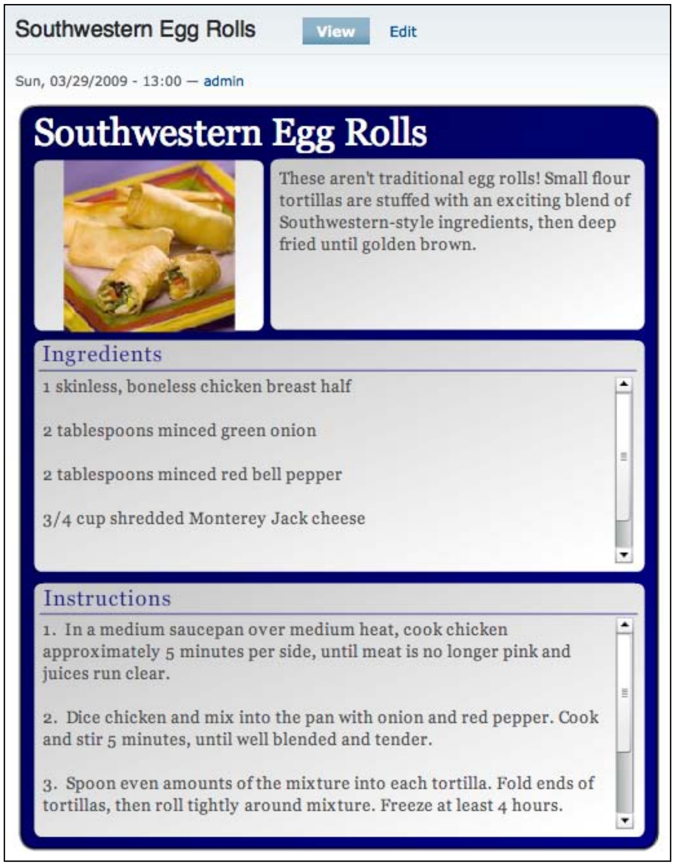
Our next step is to modify the `nodeId` variable, within the `main.as` file, back to the `FlashVar` setting, so that we can place our new Recipe application within Drupal.

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";
var nodeId:Number = root.loaderInfo.parameters.node;
```

After we run our application again, we are now ready to change our old Recipe application to our new and improved one.

Adding the new Recipe Flash application to Drupal

Since we have already done all the hard work with the node templates, this step simply involves us overwriting the old Recipe Flash application with our new one. To do this, we will first navigate to the **Administer | Content** section, where we will then locate our Flash node that we used to originally submit our Recipe Flash application (this is not the same node as the Recipe node). Once we have found the Flash node, we need to edit that node by clicking on the `edit` link next to this node. We can then change our old Flash application by simply uploading the new one using the `Flash file` input, and finally save our node when we have selected our new application. After it has been saved, we can then navigate to our Recipe node and see our new image in action!



Southwestern Egg Rolls

Sun, 03/29/2009 - 13:00 — admin

Southwestern Egg Rolls

These aren't traditional egg rolls! Small flour tortillas are stuffed with an exciting blend of Southwestern-style ingredients, then deep fried until golden brown.

Ingredients

1 skinless, boneless chicken breast half

2 tablespoons minced green onion

2 tablespoons minced red bell pepper

3/4 cup shredded Monterey Jack cheese

Instructions

1. In a medium saucepan over medium heat, cook chicken approximately 5 minutes per side, until meat is no longer pink and juices run clear.

2. Dice chicken and mix into the pan with onion and red pepper. Cook and stir 5 minutes, until well blended and tender.

3. Spoon even amounts of the mixture into each tortilla. Fold ends of tortillas, then roll tightly around mixture. Freeze at least 4 hours.

Summary

At this point, we have all the tools necessary to build some visually stunning Flash applications, where the content is provided using Drupal CMS. But, this chapter also illustrates another powerful aspect of combining Flash technology with a Content Management System. Drupal can be used as a content delivery mechanism to Flash. It can also be utilized as a manipulator of data, so that it can efficiently be streamlined into our Flash applications. The perfect example of this power is our use of the ImageCache module, where we built our Flash application to display images that were dynamically resized to fit within the Flash image region. Without a Content Management System, this functionality and integration would require a significant amount of work, whereas with Drupal, it becomes easily managed and tuned to whatever use case you may require. And the power is not in just resizing images! We can take this same concept and apply any filter imaginable (provided by ImageCache) and then show the result of those transformations within Flash to deliver a truly breathtaking experience to our users.

In this chapter, we covered all the aspects of image handling within Flash and Drupal by covering the following key topics:

- Using ImageField to load images in Drupal
- Loading those images in Flash
- Resizing images in Flash
- Retaining the aspect ratio when resizing
- Using ImageCache to dynamically resize images for our Flash applications

In the following chapter, we will take our media handling adventure to a new level by discussing how to handle audio within Flash and Drupal.

5
Drupal Audio in Flash

Within the past five years, there has been a major change in the type of content found on the World Wide Web. In just a few short years, content has evolved from being primarily text and images, into a multimedia experience! Drupal contributors have put much effort in making this integration with multimedia as easy as possible. However, one issue still remains: in order to present multimedia to your users, you cannot rely on Drupal alone. You must have another application layer to present that media. This is most typically a Flash application that allows the user to listen or watch that media from within their web browser. In the following chapters, we will cover all the necessary steps to integrate multimedia into our Drupal web site by building a custom Flash application that works seamlessly with Drupal multimedia, starting with audio content. Here is what we will learn in this chapter:

- Working with audio in Drupal
- Building a custom audio player for Drupal
- Creating an audio-handling class using ActionScript 3.0
- Loading and playing audio in Flash
- Adding controls to your custom audio player

Working with audio in Drupal

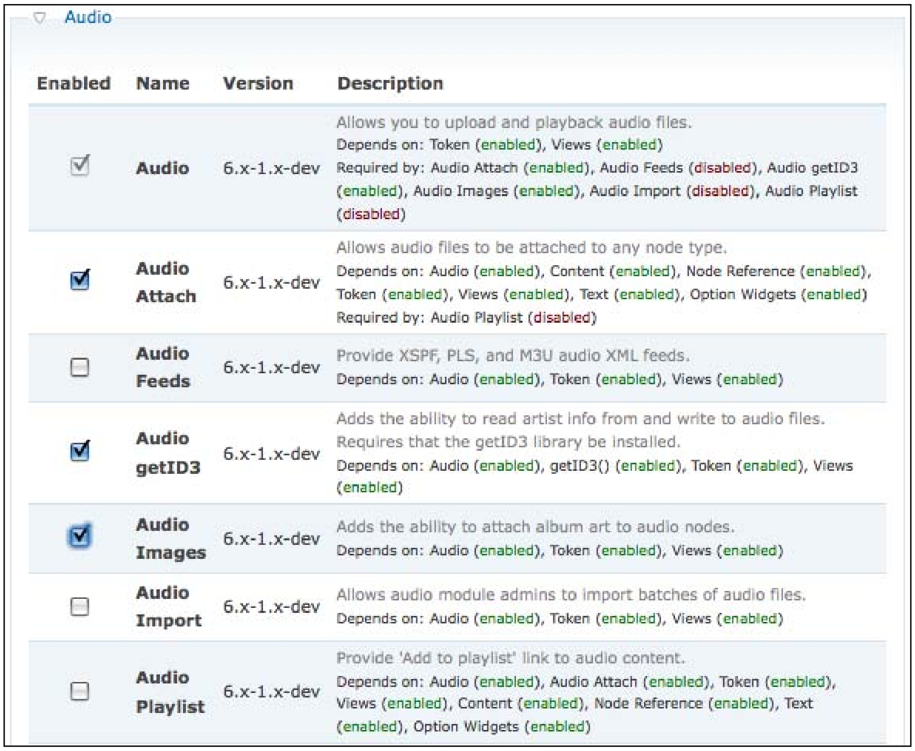
Integrating audio in Drupal is surprisingly easy, thanks to the contribution of the Audio module. This module allows you to upload audio tracks to your Drupal website (typically in MP3 format), by creating an Audio node. It also comes with a very basic audio player that will play those audio tracks in the node that was created. To start, let's download and enable the Audio module along with the Token, Views, and getID3 modules, which are required for the Audio module. The modules that you will need to download and install are as follows:

- Audio—`http://www.drupal.org/project/audio`
- Views—`http://www.drupal.org/project/views`

- Token—`http://www.drupal.org/project/token`
- getID3—`http://www.drupal.org/project/getid3`

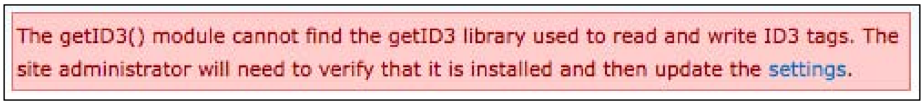
> At the time of writing this book, the Audio module was still considered "unstable". Because of this, I would recommend downloading the development version until a stable release has been made. It is also recommended to use the development or "unstable" versions for testing purposes only.

Once we have downloaded these modules and placed them in our site's `modules` folder, we can enable the **Audio** module by first navigating to the **Administer | Modules** section, and then enabling the checkboxes in the **Audio** group as follows:



Enabled	Name	Version	Description
☑	Audio	6.x-1.x-dev	Allows you to upload and playback audio files. Depends on: Token (enabled), Views (enabled) Required by: Audio Attach (enabled), Audio Feeds (disabled), Audio getID3 (enabled), Audio Images (enabled), Audio Import (disabled), Audio Playlist (disabled)
☑	Audio Attach	6.x-1.x-dev	Allows audio files to be attached to any node type. Depends on: Audio (enabled), Content (enabled), Node Reference (enabled), Token (enabled), Views (enabled), Text (enabled), Option Widgets (enabled) Required by: Audio Playlist (disabled)
□	Audio Feeds	6.x-1.x-dev	Provide XSPF, PLS, and M3U audio XML feeds. Depends on: Audio (enabled), Token (enabled), Views (enabled)
☑	Audio getID3	6.x-1.x-dev	Adds the ability to read artist info from and write to audio files. Requires that the getID3 library be installed. Depends on: Audio (enabled), getID3() (enabled), Token (enabled), Views (enabled)
☑	Audio Images	6.x-1.x-dev	Adds the ability to attach album art to audio nodes. Depends on: Audio (enabled), Token (enabled), Views (enabled)
□	Audio Import	6.x-1.x-dev	Allows audio module admins to import batches of audio files. Depends on: Audio (enabled), Token (enabled), Views (enabled)
□	Audio Playlist	6.x-1.x-dev	Provide 'Add to playlist' link to audio content. Depends on: Audio (enabled), Audio Attach (enabled), Token (enabled), Views (enabled), Content (enabled), Node Reference (enabled), Text (enabled), Option Widgets (enabled)

After you have enabled these modules, you will probably notice an error at the top of the Administrator section that says the following:

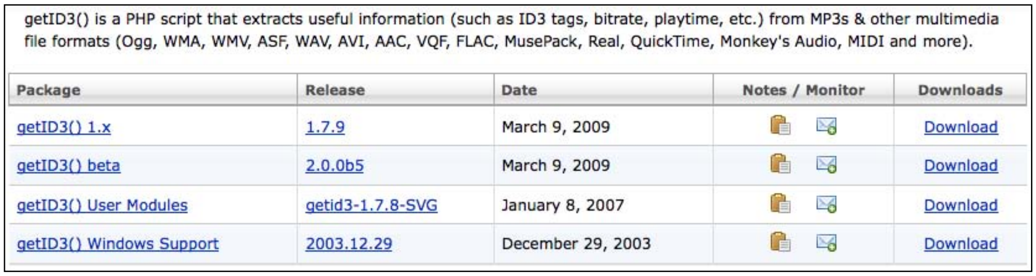


The getID3() module cannot find the getID3 library used to read and write ID3 tags. The site administrator will need to verify that it is installed and then update the settings.

This error is shown because we have not yet installed the necessary PHP library to extract the ID3 information from our audio files. The ID3 information is the track information that is embedded within each audio file, and can save us a lot of time from having to manually provide that information when attaching each audio file to our Audio nodes. So, our next step will be to install the getID3 library so that we can utilize this great feature.

Installing the getID3 library

The getID3 library is a very useful PHP library that will automatically extract audio information (called ID3) from any given audio track. We can install this useful utility by going to `http://sourceforge.net/project/showfiles.php?group_id=55859`, which is the getID3 library URL at `SourceForge.net`. Once we have done this, we should see the following:

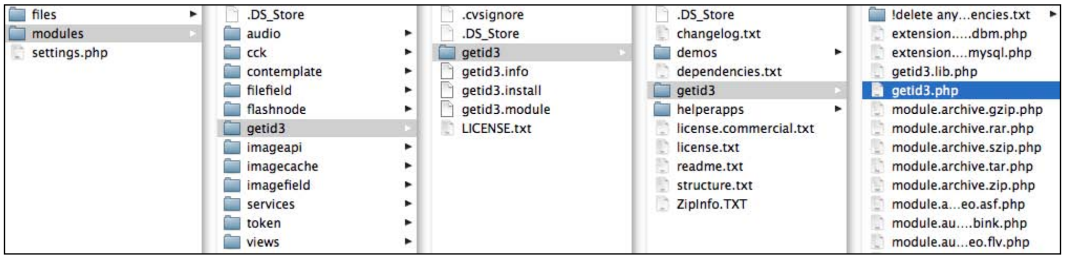


getID3() is a PHP script that extracts useful information (such as ID3 tags, bitrate, playtime, etc.) from MP3s & other multimedia file formats (Ogg, WMA, WMV, ASF, WAV, AVI, AAC, VQF, FLAC, MusePack, Real, QuickTime, Monkey's Audio, MIDI and more).

Package	Release	Date	Notes / Monitor	Downloads
getID3() 1.x	1.7.9	March 9, 2009	📋 ✉	Download
getID3() beta	2.0.0b5	March 9, 2009	📋 ✉	Download
getID3() User Modules	getid3-1.7.8-SVG	January 8, 2007	📋 ✉	Download
getID3() Windows Support	2003.12.29	December 29, 2003	📋 ✉	Download

We can download this library by clicking on the **Download** link on the first row, which is the main release. This will then take us to a new page, where we can download the ZIP package for the latest release. We can download this package by clicking on the latest ZIP link, which at the time of writing this book was **getid3-1.7.9.zip**.

Once this package has finished downloading, we then need to make sure that we place the extracted library on the server where the getID3 module can use it. The default location for the getID3 module, for this library, is within our site's `modules/getid3` directory. Within this directory, we will need to create another directory called `getid3`, and then place the `getid3` directory from the downloaded package into this directory. To verify that we have installed the library correctly, we should have the `getid3.php` at the following location:



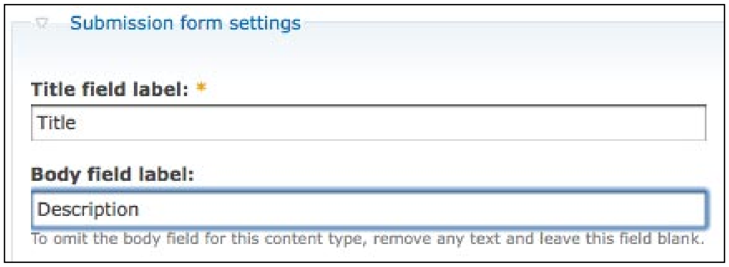
Our next task is to remove the **demos** folder from within the **getid3** library, so that we do not present any unnecessary security holes in our system.

Once this library is in the correct spot, and the **demos** folder has been removed, we can refresh our Drupal Administrator section and see that the error has disappeared. If it hasn't, then verify that your getID3 library is in the correct location and try again. Now that we have the **getID3** library installed, we are ready to set up the Audio content type.

Setting up the Audio content type

When we installed the Audio module, it automatically created an Audio content type that we can now use to add audio to our Drupal web site. But before we add any audio to our web site, let's take a few minutes to set up the Audio content type to the way we want it. We will do so by navigating to **Administer | Content Types**, and then clicking on the **edit** link, next to the **Audio** content type.

Our goal here is to set up the Audio content type so that the default fields make sense to the Audio content type. Much like the Recipe node type that we created in earlier chapters, Drupal adds the **Body** field to all new content types, which doesn't make much sense when creating an Audio content. We can easily change this by simply expanding the **Submission form settings**. We can then replace the **Body** label with **Description**, since it is easily understood when adding new Audio tracks to our system.



▽ Submission form settings

Title field label: *

Title

Body field label:

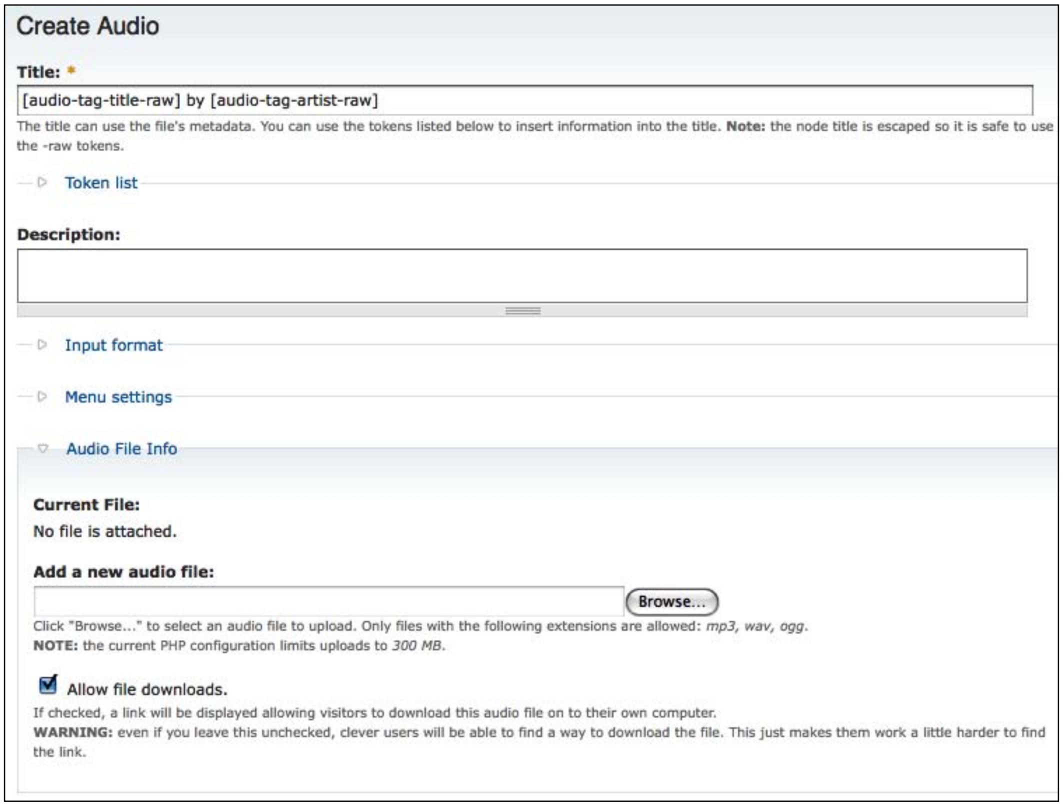
Description

To omit the body field for this content type, remove any text and leave this field blank.

We will save this content type by clicking on the **Save content type** button at the bottom of the page. Now, we are ready to start adding audio content to our Drupal web site.

Creating an Audio node

We will add audio content by going to **Create Content,** and then clicking on **Audio**, where we should then see the following on the page:



Create Audio

Title: *

[audio-tag-title-raw] by [audio-tag-artist-raw]

The title can use the file's metadata. You can use the tokens listed below to insert information into the title. **Note:** the node title is escaped so it is safe to use the -raw tokens.

▷ Token list

Description:

▷ Input format

▷ Menu settings

▽ Audio File Info

Current File:

No file is attached.

Add a new audio file:

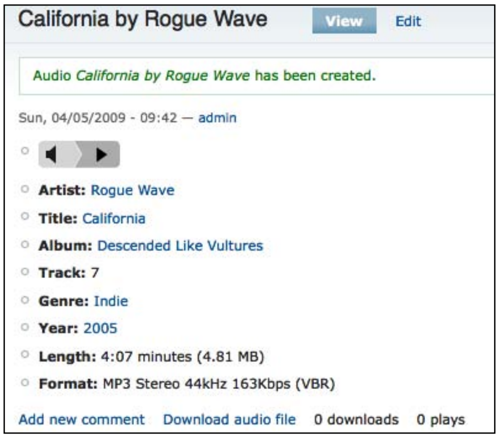
Browse...

Click "Browse..." to select an audio file to upload. Only files with the following extensions are allowed: *mp3, wav, ogg*.
NOTE: the current PHP configuration limits uploads to *300 MB*.

☑ Allow file downloads.

If checked, a link will be displayed allowing visitors to download this audio file on to their own computer.
WARNING: even if you leave this unchecked, clever users will be able to find a way to download the file. This just makes them work a little harder to find the link.

You will probably notice that the **Title** of this form has already been filled out with some strange looking text (as shown in the previous screenshot). This text is a series of tags, which are used to represent track information that is extracted using the getID3 module that we installed earlier. Once this ID3 information is extracted, these tags will be replaced with the **Title** and **Artist** of that track, and then combined to form the title of this node. This will save a lot of time because we do not have to manually provide this information when submitting a new audio track to our site. We can now upload any audio track by clicking on the **Browse** button next to the **Add a new audio file** field. After it adds the file to the field, we can submit this audio track to Drupal by clicking on the **Save** button at the bottom of the page, which will then show you something like the following screenshot:



Audio *California by Rogue Wave* has been created.

Sun, 04/05/2009 - 09:42 — admin

- **Artist:** Rogue Wave
- **Title:** California
- **Album:** Descended Like Vultures
- **Track:** 7
- **Genre:** Indie
- **Year:** 2005
- **Length:** 4:07 minutes (4.81 MB)
- **Format:** MP3 Stereo 44kHz 163Kbps (VBR)

Add new comment Download audio file 0 downloads 0 plays

After this node has been added, you will notice that there is a player already provided to play the audio track. Although this player is really cool, there are some key differences between the player provided by the Audio module and the player that we will create later in this chapter.

How our player will be different (and better)

The main difference between the player that is provided by the Audio module and the player that we are getting ready to build is how it determines which file to play. In the default player, it uses flash variables passed to the player to determine which file to play. This type of player-web site interaction places the burden on Drupal to provide the file that needs to be played. In a way, the default player is passive, where it does nothing unless someone tells it to do something.

The player that we will be building is different because instead of Drupal telling our player what to play, we will take an active approach and query Drupal for the file we wish to play. This has several benefits, such as that the file path does not have to be exposed to the public in order for it to be played. So, let's create our custom player!

Building a custom audio player for Drupal

In this section we will expand the Flash project from Chapter 2, and use that to create a custom audio player for Drupal. So, let's get started by copying the `chapter2` directory, and then create a new directory called `chapter5` that we will use to keep track of all of our changes. After we have done that, we should then rename the `chapter2.fla` project file to `chapter5.fla`. Once we have our new directory set up, we will need to open up both the `chapter5.fla` and the `main.as` file within our Flash IDE, where we will then direct our attention once again to the `main.as` file.

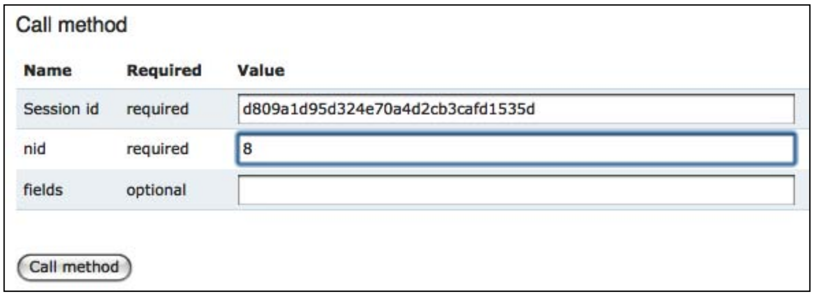
The first thing we will need to do is temporarily change the `nodeId` variable at the top of this script to the node ID of the audio node that we just created as follows:

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";
var nodeId:Number = 8;
```

Now that this node ID is set to the correct node, our next task is to determine what data we are looking for when we load the node. This will bring us back to our Drupal web site, where we will take advantage of the Services Administrator to investigate the data from our audio node.

Examining the Audio node using Services Administrator

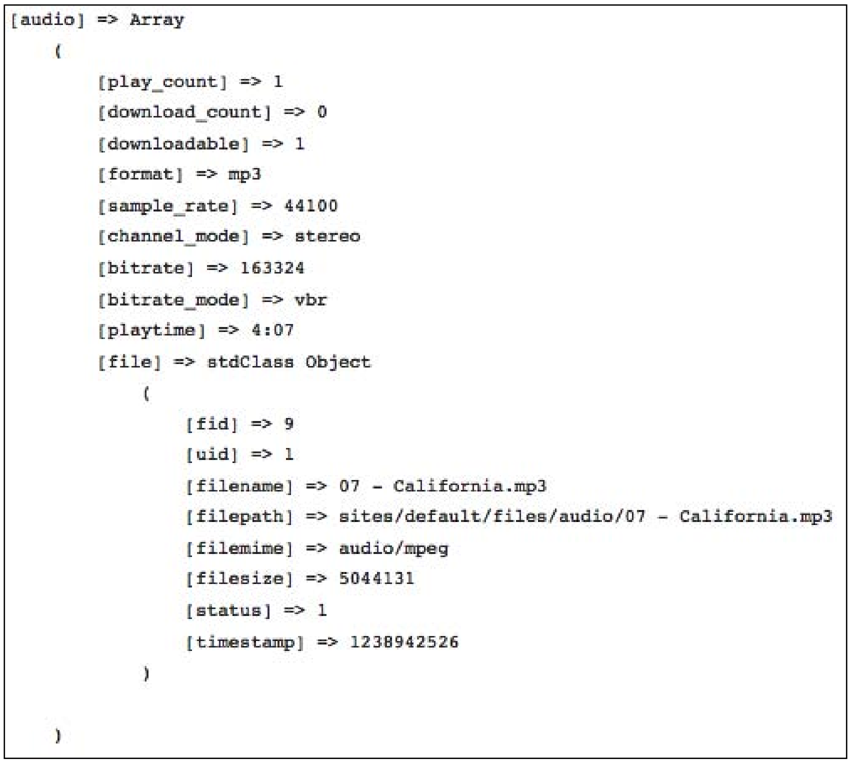
For this section, we will navigate back to our Services Administrator section by going to **Administer | Services** in our Drupal web site. Once we are there, we will then click on the **node.get** link, which will let us load any node in our system to examine the data that will be passed to our Flash Application. We will then need to provide the node ID for the audio node we created—where it says **nid**, and then click on the button below that says **Call method**.



Call method

Name	Required	Value
Session id	required	d809a1d95d324e70a4d2cb3cafd1535d
nid	required	8
fields	optional	

Looking at the results from this call, the data that we are looking for is all contained within the **audio** tag in the node object, which should look similar to the following screenshot:



```
[audio] => Array
    (
        [play_count] => 1
        [download_count] => 0
        [downloadable] => 1
        [format] => mp3
        [sample_rate] => 44100
        [channel_mode] => stereo
        [bitrate] => 163324
        [bitrate_mode] => vbr
        [playtime] => 4:07
        [file] => stdClass Object
            (
                [fid] => 9
                [uid] => 1
                [filename] => 07 - California.mp3
                [filepath] => sites/default/files/audio/07 - California.mp3
                [filemime] => audio/mpeg
                [filesize] => 5044131
                [status] => 1
                [timestamp] => 1238942526
            )

    )
```

From looking at this data structure, we determine that we can access the filepath of our audio node within our `onNodeLoad` function. So, let's test this out by modifying our "Hello World" code to replace the node title with the filepath to our audio file.

Referencing the audio file path

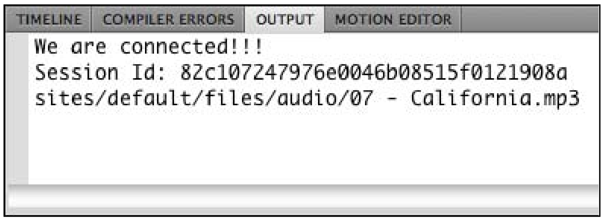
Using the knowledge that we gained from the Services Administrator, we should be able to now reference the audio filepath for any given audio node within our Drupal web site. If we observe the node object data returned from our Services Administrator, we can determine how to access the file path to our song by using the following code:

```
node.audio.file.filepath
```

We can easily test this out by opening up our `main.as` file, and then placing a trace statement to display this file path within the `onNodeLoad` function:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;

   // Trace the audio file path.
   trace( node.audio.file.filepath );
}
```

The output should then show the correct filepath to the node that we just loaded (as shown in the following screenshot).



```
We are connected!!!
Session Id: 82c107247976e0046b08515f0121908a
sites/default/files/audio/07 - California.mp3
```

We have now successfully referenced the audio file path. Our next task will be to create an audio class that will use this path to play some music!

Writing a custom AudioPlayer class

When working with ActionScript 3, it is highly recommended to use the object-oriented features that are built into the language using the `class` construct. By creating a class for our custom audio functionality, we will be encapsulating the code, which makes our code more maintainable, expandable, and portable. This is also referred to as componentization. This section assumes that you already have some previous experience with object-oriented techniques, but in case you do not, I will try my best to explain the concepts as we move forward. If you are just beginning with object-oriented programming, then I would also highly recommend reading the Wikipedia article at `http://en.wikipedia.org/wiki/Object-oriented_programming`, which describes in great detail the concepts behind object-oriented programming. With that said, let's begin building our `AudioPlayer` class.

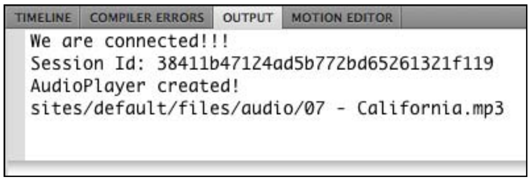
In Flash, there is already a class called `Sound` that was built to play audio files, and we can build our class to utilize this functionality to play audio. So, let's begin by creating a blank file next to your `chapter5.fla` project file called `AudioPlayer.as`. We will then open up this file and write the following:

```
package
{
   // Import all dependencies
   import flash.media.Sound;
    // Declare our class
   public class AudioPlayer
   {
   // Constructor function.
   // Called when someone creates a new AudioPlayer
      public function AudioPlayer()
      {
         // Make sure to create our sound object
         sound = new Sound();
         // Let us know that we created this player.
         trace( "AudioPlayer created!" );
      }
      // Declare our sound variable.
      private var sound:Sound;
   }
}
```

Here we have created a new class that we will use to place all of our custom Audio player functionality. Currently, this doesn't really do much, other than send a trace to the output to notify us that the player has been created. To help track our progress, we can test this out by going back to our `main.as` file, and within the `onNodeLoad` function, we can place the following code to create our custom `AudioPlayer`:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;

   // Create our AudioPlayer.
   var player:AudioPlayer = new AudioPlayer();
   // Trace the audio file path.
   trace( node.audio.file.filepath );
}
```

Now, when we run this application, we will get a very pleasant surprise when our trace statement from within our custom AudioPlayer gets called to reveal that we really did create our custom audio player!



```
We are connected!!!
Session Id: 38411b47124ad5b772bd65261321f119
AudioPlayer created!
sites/default/files/audio/07 - California.mp3
```

Our next step will be to add functionality to our custom audio class to play any audio track passed to our player.

Playing audio in Flash

In order to play an audio track, we will need to first create a public function within our custom class, that will be used to play any given file passed to our routine.

This function will be used to play any given audio file path provided as the file string passed as an argument to the play function. Since we have already included the sound object in our custom class, we can now use that to load and play our file.

To do this, we will need to import the `URLRequest` class, because that class is used to pass a URL string to the load routine of the sound object. After this, we can then call the load routine on the sound object using this `URLRequest` object, and then play the file after it has been loaded. This will look as follows:

```
package
{
   // Import all dependencies
   import flash.media.Sound;
   import flash.net.URLRequest;
    // Declare our class
   public function AudioPlayer
   {
      // Constructor function.
      // Called when someone creates a new AudioPlayer
      public function AudioPlayer()
      {
         // Make sure to create our sound object
         sound = new Sound();
         // Let us know that we created this player.
         trace( "AudioPlayer created!" );
      }
      // Play an audio file
      public function playFile( file:String )
      {
         // Print out what file is playing...
         trace( "Playing file " + file );
         // Load our sound file.
         sound.load( new URLRequest( file ) );
         // Play our sound file.
         sound.play();
      }
      // Declare our sound variable.
      private var sound:Sound;
   }
}
```

We have finished setting up our audio class to play an audio file. We can now direct our attention to the main.as file, where we will play the audio file from Drupal using our new custom audio class.

Using our AudioPlayer class to play audio

Now that we have our main.as file opened, we can direct our attention once again to the onNodeLoad function, where we will pass the correct file path from Drupal to our custom AudioPlayer class. Since the path to our audio file, given to us from the node object, is relative to the base URL of our Drupal web site, we will need to do the same thing that we did in the previous chapter, where we added the base URL of our web site to the front of this path before we send it to the play function of our custom class. We can do this pretty easily by creating a variable called fileURL, which will hold the baseURL to our web site, and then add that to the audio file path before sending it to the play function of our custom class. The code to do this should look like the following:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
   // Create our AudioPlayer.
   var player:AudioPlayer = new AudioPlayer();
   // Declare our base URL.
   var fileURL:String = baseURL;
   // Add our file's relative path.
   fileURL += "/";
   fileURL += node.audio.file.filepath;
   // Play our audio file
   player.playFile( fileURL );
}
```

Now, when we run our application, we should be greeted with the sweet sound of success!

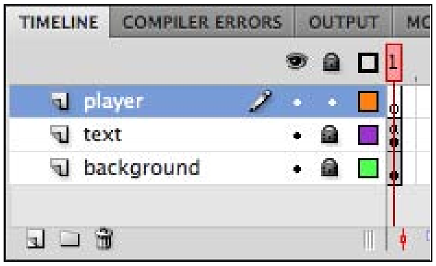
We will now expand our audio player to include some controls, so that your site visitors can start and stop the music while playing.

Adding controls to your custom audio player

Playing music by itself is pretty cool, but is not very useful unless we give our users a way to interact with the playback of that audio track. In this section we will create some very basic controls that will allow our users to do just that. Although there are a handful of controls that we can add to this custom audio player, this section will demonstrate the concept by adding the most basic control for multimedia, which is the play and pause buttons.

Adding a play and pause button

To begin, we will need to first move and resize our title field within our Flash application, so that it can hold more text than "Hello World". We can then make room for some new controls that will be used to control the playback of our audio file. Again, the design of each of these components is subjective, but what is important is the MovieClip instance hierarchy, which will be used within our ActionScript code. Before we begin, we will need to create a new layer in our **TIMELINE** that will be used to place all `AudioPlayer` objects. We will call this new layer **player**:



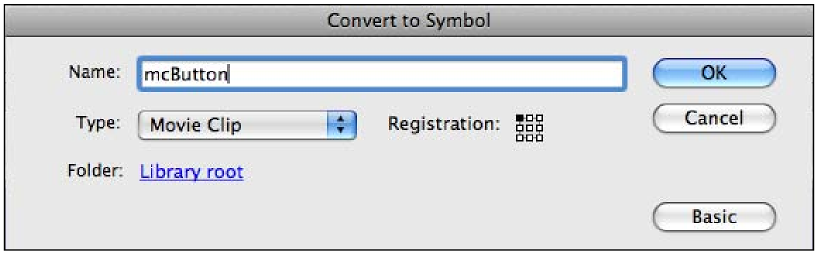
We can now proceed to creating our play and pause buttons.

Creating a base button MovieClip

Our base button will simply be a rounded rectangle, which we will then add some gradients to, so as to give it depth. We can do this by first creating a rounded rectangle with a vertical linear gradient fill as follows:

We can now give it some very cool depth by adding a smaller rounded rectangle within this one, and then orient the same gradient horizontally. An easy way to do this is to copy the original shape and paste it as a new shape. Once we have a new copy of our original rounded rectangle, we can navigate to **Modify | Shape | Expand fill**, where we will then select **Inset**, change our **Distance** to 4px, and then click on **OK**. After doing this, you will realize how such a simple contrast in gradients can really bring out the shape.

After we have our new button shape, we will then need to create a new MovieClip, so that we can reuse this button for both the play and pause buttons. To do this, simply select both the rounded rectangle regions, and then choose **Modify | Convert to Symbol** in the Flash menu. We are going to call this new movie clip **mcButton**.



Convert to Symbol

Name: mcButton

Type: Movie Clip

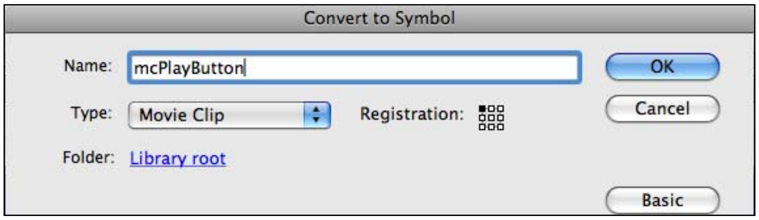
Registration:

Folder: Library root

Now that we have a base button MovieClip, we can now add the play and pause symbols to complete the play and pause buttons.

Adding the PlayButton movie clip

The first button that we will create is the play button, which simply consists of a sideways triangle (icon) with the button behind it. To do this, we will first create a new movie clip that will hold the button we just created, and the play icon. We can do this by first clicking on the **mcButton** movie clip, and then creating a new movie clip from that by selecting **Modify | Convert to Symbol**. We will call our new movie clip **mcPlayButton**.


Convert to Symbol

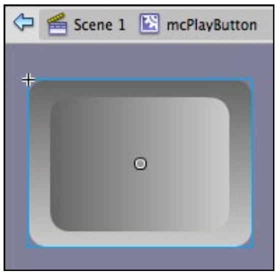
Name: mcPlayButton

Type: Movie Clip

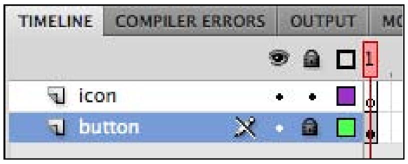
Registration:

Folder: Library root

What we are really doing here is creating a parent movie clip for our **mcButton**, which will allow us to add new specific elements. For the play button, we simply want to add a play symbol. To do this, we first want to make sure that we are within the **mcPlayButton** movie clip by double-clicking on this symbol, so that our breadcrumb at the top of the stage looks as follows:



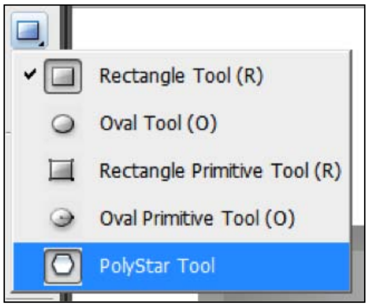
Our next task is to modify our timeline within this movie clip so that we can separate the icon from the button. We can do this by creating two new layers within our timeline, called **button** (which will hold our button) and **icon** (which we will create in the next section).



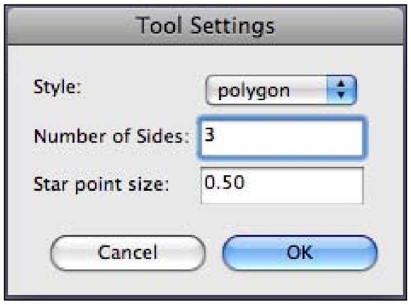
We are now ready to start drawing the play icon.

Drawing a play icon

To draw a Play icon, we will need to first select the **PolyStar Tool** by clicking and holding on the □ tool until you can select the **PolyStar Tool**.



This tool will allow us to create a triangle, which we will use for the play icon in our play button. But before we can start drawing, we need to first set up the **PolyStar Tool** so that it will draw a triangle. We can do this by clicking on the **Options** button within the **Properties** tab, which will then bring up a dialog, where we can tell it to draw a polygon with three sides (triangle).



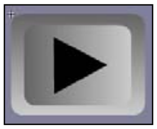
Style: polygon

Number of Sides: 3

Star point size: 0.50

After we click on **OK**, we will then need to change the fill color of this triangle, so that it is visible on our button. We will just change the fill color to Black. ■

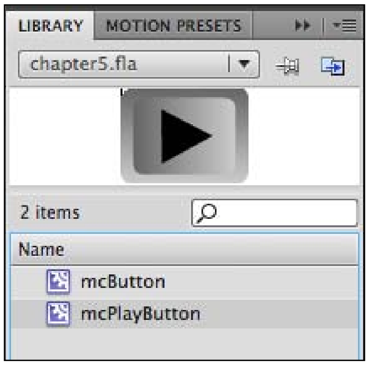
We can then move our cursor onto the stage where the button is, and then draw our triangle in the shape of a play button icon. Remember, if you do not like the shape of what you made, you can always tweak it using the transform ⬚ tool. When we are done, we should have something that resembles a play button!



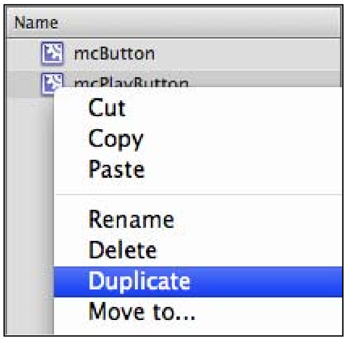
Our next task is to create a pause button. Since we have already created the play button, which is similar to the pause button except for the icon, we can use a handy tool in Flash that will let us duplicate our play button, and then modify our duplication for the pause button icon.

Creating a pause button from the play button

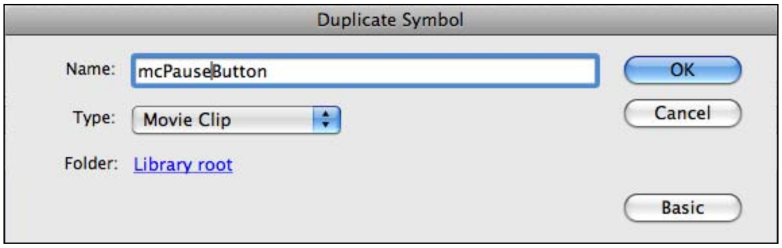
In order to create our pause button, we will first need to duplicate our play button into a new movie clip, where we can change the icon from play to pause. To do this, we will first direct our attention to the library section of our Flash IDE, which should show us all of the movie clips that we have created so far. We can find the **LIBRARY** by clicking on the 📚 button on the right-hand side of our workspace.



To create a duplicate, we will now right-click on the **mcPlayButton** movie clip, and then select the option **Duplicate**.



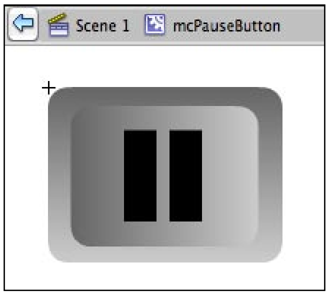
This will then bring up a dialog very similar to the dialog when we created new symbols, but this time, we are defining a new movie clip name that will serve as a duplicate for the original one. We will call our new movie clip duplicate **mcPauseButton**.



Name: mcPauseButton

Type: Movie Clip

Now that we have created our duplicate movie clip, the next task is to change the icon within the pause button. We can do this by opening up our **mcPauseButton** movie clip by double-clicking on that name within the **Library**. At this point, we can now change the icon of our pause button without running any risk of also modifying the play button (since we created a duplicate). When we are done, we should have a complete pause button.

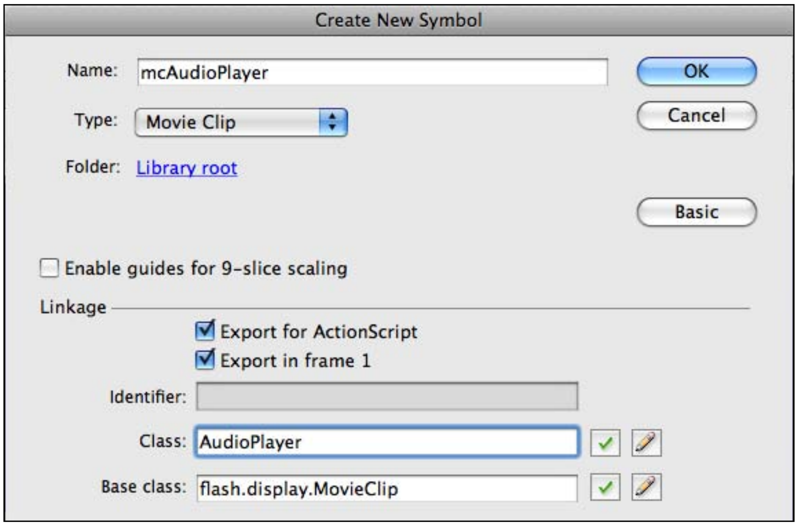


We now have play and pause buttons that we will use to link to our `AudioPlayer` class.

Linking MovieClips to ActionScript

One overlooked feature within the Flash IDE is its ability to link MovieClips to ActionScript code, which is used to add scripted functionality to movie clips. This is an extremely powerful feature that we will use to create the play and pause buttons automatically when we create the `AudioPlayer` object. To start, we will need to combine both the **mcPlayButton** and the **mcPauseButton** into a single MovieClip that will directly link to the `AudioPlayer` class, which we created earlier in this chapter. This movie clip that we will create will be called **mcAudioPlayer** since it is directly representative of the class that will govern it. With that said, we will now create a new MovieClip by first navigating back to our stage, and then clicking on the **Insert | New Symbol** from our Flash IDE.

Now, before we do anything, I would like to point out a difference in how we will create this MovieClip versus other MovieClips that we have created in the past. This time, instead of just giving our MovieClip a name (**mcAudioPlayer**), we will also check the box that says **Export for ActionScript**. What this is doing is linking this movie clip to ActionScript code, and even cooler, we can provide a class that will link its functionality to this MovieClip by filling out the class name where it says **Class**. Before hitting **OK**, make sure that your MovieClip definition looks like the following:


Create New Symbol

Name: mcAudioPlayer

Type: Movie Clip

Folder: Library root

□ Enable guides for 9-slice scaling

Linkage

☑ Export for ActionScript

☑ Export in frame 1

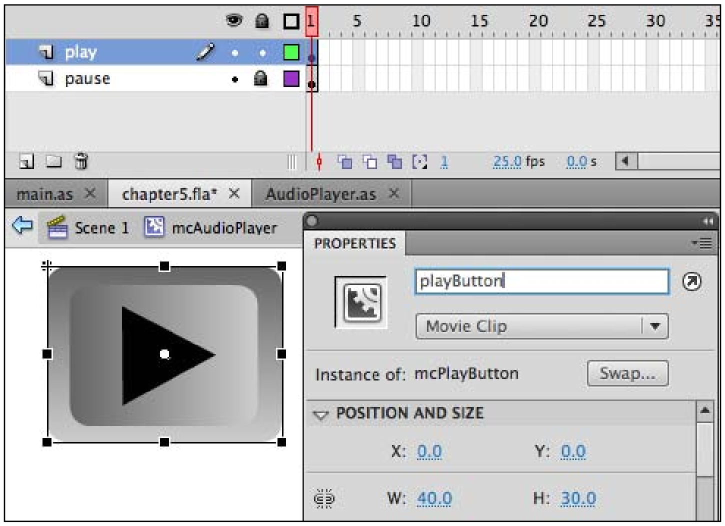
Identifier:

Class: AudioPlayer

Base class: flash.display.MovieClip

After we have done this, it will open up our new movie clip for editing, where we will add both the **mcPlayButton** and the **mcPauseButton**. Again, it is highly recommended to place each of these MoveClips on their own separate layer within the **mcAudioPlayer** movie clip. We will also need to make sure that both the X and Y positions for both the **mcPlayButton** and **mcPauseButton** are 0. It is also important to give each of our buttons an instance name using the **Properties** panel for each movie clip. This is so that we can reference them within ActionScript. We will call our play and pause buttons **playButton** and **pauseButton** respectively.

When we are done, our **mcAudioPlayer** MovieClip should look like the following on your stage:



playButton

Movie Clip

Instance of: mcPlayButton

X: 0.0 Y: 0.0

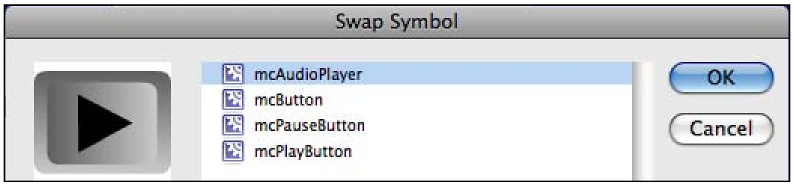
W: 40.0 H: 30.0

Our next and final task is to add our new **mcAudioPlayer** movie clip to our stage.

Adding the AudioPlayer to the stage

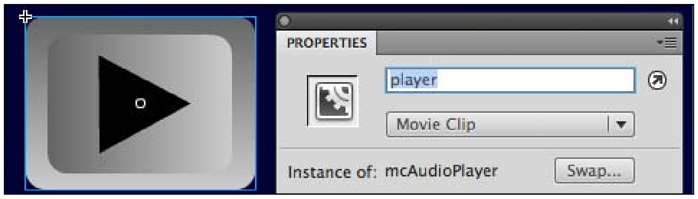
If we were to navigate back to the stage, we will see that the **mcPlayButton** movie clip is currently being shown on the stage. Because of this, we will need to swap this object out for the **mcAudioPlayer** movie clip, since it now contains both the play and pause buttons. Luckily, there is a very handy operation in Flash that allows for us to swap one symbol with another.

When we click on the **mcPlayButton** movie clip, and look over at the **Properties** panel, we should see a button called **Swap** that will allow us to switch one movie clip for another. This is perfect for our use case since we would like to swap the **mcPlayButton** movie clip with the **mcAudioPlayer** movie clip that we just finished creating. When we click on the **Swap** button, we are presented with a list of movie clips that we would like to swap. In this list, we can select the **mcAudioPlayer** and then click on **OK** to accept the swap.



mcAudioPlayer
mcButton
mcPauseButton
mcPlayButton

We can now give our new **mcAudioPlayer** movie clip an instance so that it can be referenced within our ActionScript code. We will call it **player**.



Once we do this, we will be ready to make some modifications to our `AudioPlayer` class to reference the **mcAudioPlayer** movie clip that we just created.

Modifying the AudioPlayer class to use play and pause

In this section, we will take our `AudioPlayer` class and modify it so that it will work with the **mcAudioPlayer** movie clip that we just created, which incorporates the play and pause buttons. We can accomplish this with a series of quick steps.

Step 1: Adding the SoundChannel

Before we begin, we will need to add the mechanism to our `Sound` class that allows us to control the audio channel that is currently being played. In Flash, there is a class that we will use to do this called `SoundChannel`. We can use the `SoundChannel` as a member variable with sound that will keep track of the current track position. Because of this, we will also need to add another variable called `position` to our `AudioPlayer` class. These variables will live along with the sound variable that we already created. We will also need to include the dependency for the `SoundChannel` variable, which we can place at the top of the class.

```
// Import all dependencies
import flash.media.Sound;
import flash.media.SoundChannel;
import flash.net.URLRequest;
...
...
...
// Declare our sound variable.
private var sound:Sound;
// Declare our sound channel
private var channel:SoundChannel;
// Variable to keep track of the audio position.
private var position:Number;
```

Step 2: Adding load, play, and pause functions

The next major change that we will need to make is to modify our class so that we can load, play, and pause the file in three separate functions. This will require us to completely change the current play function that we have, so that it matches this new structure. The `play` function will set the audio channel and play the file at the current position. The `pause` function will first save the channel position and then stop the audio file. And finally, the load function will simply load the file passed to our `AudioPlayer` class. These changes are as follows:

```
// Constructor function.
// Called when someone creates a new AudioPlayer
public function AudioPlayer()
{
   // Make sure to create our sound object
   sound = new Sound();
   // Initialize the position.
   position = 0;
   // Let us know that we created this player.
   trace( "AudioPlayer created!" );
}

// Play an audio file
public function playFile()
{
   // Play our sound file.
   channel = sound.play(position);
}

// Pause an audio file
public function pause()
{
   // Save the channel position.
   position = channel.position;
   // Stop our sound file.
   channel.stop();
}

// Load an audio file.
public function load( file:String )
{
   // Load our sound file.
   sound.load( new URLRequest( file ) );
}
```

Step 3: Reference the mcAudioPlayer MovieClip

After we have done that, the next task will be to change our class so that it officially references the **mcAudioPlayer** movie clip that we created. We can do this by simply stating that our class derives from the MovieClip class. This can be done using the `extends` keyword, when we declare our class as follows:

```
// Import all dependencies
import flash.display.MovieClip;
import flash.media.Sound;
import flash.media.SoundChannel;
import flash.net.URLRequest;
// Declare our class
public class AudioPlayer extends MovieClip
{
   // Constructor function
   // Called when someone creates a new AudioPlayer
   public function AudioPlayer()
   // Make sure we call the MovieClip constructor
      super();

   // Make sure to create our sound object
   sound = new Sound();
   // Initialize the position.
   position = 0;

   // Let us know that we created this player.
   trace( "AudioPlayer created!" );
}
```

Step 4: Hooking up our buttons!

The last and final step in creating our `AudioPlayer` class is to hook up the play and pause buttons that we added to our **mcAudioPlayer** movie clip object. Since we already told Flash to reference the `AudioPlayer` class when creating **mcAudioPlayer** movie clip, the instance names for the play and pause buttons can now be used within our class to manipulate their behavior during audio playback. These instance names were `playButton` and `pauseButton`, and we will start out by declaring them as buttons.

Declaring playButton and pauseButton as buttons

We will declare `playButton` and `pauseButton` as buttons using the `buttonMode` parameter along with an event handler that will call any given function when that button is clicked. The `mouseChildren` parameter is used to tell Flash to not let any child movie clip within these buttons to get focus. This functionality will be placed within the `load` function of our class, and then we will set their visibility state according to which button should be shown (which we will default as the play button).

```
// Import all dependencies
import flash.display.MovieClip;
import flash.media.Sound;
import flash.media.SoundChannel;
import flash.net.URLRequest;
import flash.events.MouseEvent;
...
...
...
// Load an audio file.
public function load( file:String )
{
   // Setup the play button.
   playButton.buttonMode = true;
   playButton.mouseChildren = false;
   playButton.addEventListener(MouseEvent.MOUSE_UP,onPlay);
   // Setup the pause button.
   pauseButton.buttonMode = true;
   pauseButton.mouseChildren = false;
   pauseButton.addEventListener(MouseEvent.MOUSE_UP,onPause);
   // Set the state of the play and pause buttons.
   // We want to show the play button at first, so...
   playButton.visible = true;
   pauseButton.visible = false;
   // Load our sound file.
   sound.load( new URLRequest( file ) );
}
```

Along with this, we need to create two handler functions that will handle the mouse events and then make the correct function calls depending on which button the user clicks.

```
// Load an audio file.
public function load( file:String )
{
   // Setup the play button.
   playButton.buttonMode = true;
   playButton.mouseChildren = false;
   playButton.addEventListener(MouseEvent.MOUSE_UP,onPlay);
   // Setup the pause button.
   pauseButton.buttonMode = true;
   pauseButton.mouseChildren = false;
   pauseButton.addEventListener(MouseEvent.MOUSE_UP,onPause);

   // Set the state of the play and pause buttons.
   // We want to show the play button at first, so...
   playButton.visible = true;
   pauseButton.visible = false;

   // Load our sound file.
   sound.load( new URLRequest( file ) );
}
// Called when the play button has been pressed.
private function onPlay( event:MouseEvent )
{
   // Play the audio track.
   playFile();
}
// Called when the user presses the pause button.
private function onPause( event:MouseEvent )
{
   // Pause the audio track.
   pause();
}
```

Now, all we need to do is make sure we change the state of these buttons as they are clicked, by adding the following code to the play and pause functions:

```
// Play an audio file
public function playFile()
{
   // Play our sound file.
   channel = sound.play(position);
   // Show only the pause button.
```

```
      playButton.visible = false;
      pauseButton.visible = true;
   }
   // Pause an audio file
   public function pause()
   {
      // Save the channel position.
      position = channel.position;
      // Stop our sound file.
      channel.stop();
      // Show only the pause button.
      playButton.visible = true;
      pauseButton.visible = false;
   }
```

We have now finished making modifications to our AudioPlayer class to allow the play and pause buttons to work.

The last and final step is to make two very minor modifications to our main audioplayer.fla project to account for the changes that we have made.

Modifying our main.as file to use our new AudioPlayer

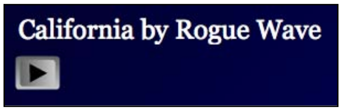
Now, moving back to the main.as file, we will make some very simple modifications to our ActionScript code to allow the AudioPlayer to work the way we want it to. The first change that we will need to make is to remove our call to create the new player using new AudioPlayer(). The reason we can remove this is because Flash has done this for us when we added the mcAudioPlayer movie clip to our stage and then gave it an instance name of player. Because of this, we now have a valid AudioPlayer on our stage when the Flash movie is created, and we can then use the player instance to reference that AudioPlayer.

The last and final change that we will need to make is to change the play call from the player to the load call, which will set up our AudioPlayer's buttons to allow the user to control whether the audio should play or not. Our new onNodeLoad function should look like the following:

```
   // Called when Drupal returns with our node.
   function onNodeLoad( node:Object )
   {
      // Print out the node title.
      title.text = node.title;
       // Declare our base URL.
      var fileURL:String = baseURL;
```

```
        // Add our file's relative path.
        fileURL += "/";
        fileURL += node.audio.file.filepath;
        // Load our audio file
        player.load( fileURL );
    }
```

When we run this, we will now be happy to see that our play/pause button works as expected, and we are now able to control the play or pause state of the audio track being played!


California by Rogue Wave

A good exercise from here would be to take the lessons learned from the previous chapters, and then build our audio player for a live site (by changing the `nodeId` variable at the top of the page to use FlashVars). Once we are done with that, we can then replace the default theme for the Audio node type and use our new player to play the audio tracks attached to each audio node!

Summary

In this chapter we learned how audio is handled within Drupal and how to build a custom application that can play and pause audio content created through Drupal. There are several key points that I would like you to remember as you read through to the next chapters:

- The Audio module is a contributed module that allows Drupal to upload and automatically tag audio content using the getID3 library.
- We can determine how to reference that audio content in Flash using Drupal's Services Administrator section.
- We designed our custom audio player to take advantage of the object-oriented techniques provided by ActionScript 3 using the class construct. By doing this, we allowed for our implementation to be easily maintained, portable, and expanded.
- When creating the **mcAudioPlayer** movie clip, we were able to link the ActionScript code to the MovieClip buttons that we created using the **Export to ActionScript** checkbox.

In the next chapter we will take our implementation of Drupal multimedia one step further by discussing how to handle Video content using Drupal and Flash.

6
Flash Video in Drupal

In recent years the amount of bandwidth available to the masses has resulted in a surge of video content. As a result of this surge, the integration of video and Content Management Systems has become a necessary step to help manage and deliver that video content to millions of viewers every day. In this chapter we will learn how Drupal handles video content, and how to build a custom video player to interface with that content. We'll cover the following key topics:

- Working with video in Drupal
- Learning how to utilize object-oriented techniques in ActionScript 3.0 to build a common MediaPlayer class to be used for both audio and video applications
- Building a custom video player in Flash
- Linking our custom video player to the Drupal content
- Dynamically selecting which player to use depending on node media content
- Adding our custom media player to Drupal

Working with video in Drupal

Before we dive into Flash, we will first explore the process of implementing video into our Drupal web site. Much like audio and images, the Drupal contributors have introduced numerous modules for integrating video into our web site. Consequently, the process of finding the right solution for your needs can be somewhat confusing, since each one was developed to handle specific video requirements. In this section, we will briefly explore the different video solutions available for video integration, and then choose one that best fits our needs in this chapter. It is important to note, however, that the video solution we choose may not be the only or the best solution, but simply a solution that fits the bill for what we need to get done.

Although it is not an exhaustive list, following are four different video modules available for Drupal 6 at the time of writing this book. Each one of these modules was designed with a different use case in mind.

- **Embedded Media Field** (`http://www.drupal.org/project/emfield`)—this is a fantastic module, written to integrate third-party videos into your site such as YouTube, Brightcove, and many more. It works as a plug-in for CCK to allow any content type to attach a third-party video to that node.
- **FlashVideo** (`http://www.drupal.org/project/flashvideo`)—this module is used as a standalone video module that gives Drupal the power to host and maintain all of its own video content, without relying on third-party video web sites. It does this by providing an upload and conversion mechanism that gives Drupal the ability to host and maintain its own user-generated Flash video content. This solution is ideal to site owners wanting a user-generated video content system, without relying on third-party web sites to host and maintain those videos.
- **Media Mover** (`http://www.drupal.org/project/media_mover`)—this is a fantastic video module geared more towards the administrative management of video content on a Drupal web site. As far as the administrative abilities for video management are concerned, this module is top of its class. It gives the administrators the ability to specify where their videos are coming from and where they are going, as well as any conversion that needs to be handled in between.
- **FileField + jQuery Media** (`http://www.drupal.org/project/jquery_media`)—these two modules work as a great team to allow an individual to upload a pre-converted video to use on their Drupal web site. This solution is perfect for web sites where the users do not provide video content on the site, but the videos are uploaded by the site administrators who have the capability to convert the videos to Flash format before they upload them to the site.

Given these four different video modules and their specific use case, we can now select the one that is appropriate for this chapter. Since this chapter will mostly involve us creating our very own video player, I think it is wise to choose the solution that allows us to upload pre-converted videos that we will then show in our custom player. For this reason, we will proceed with using the **FileField + jQuery Media** video solution.

But, before we begin installing the necessary modules for video functionality, we first need to create a new content type that will be used to hold all of our video content.

Creating a video content type

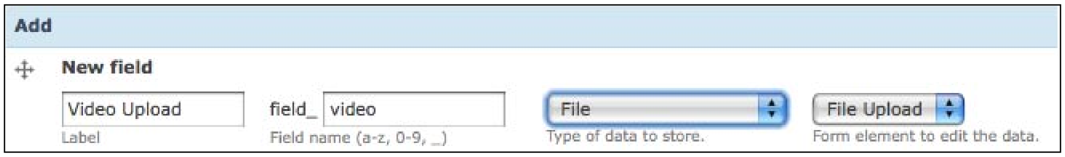
In this section we will use the same methods from the previous chapters to create a new content type called **video**. To do this, we will need to navigate to the **Administer | Content Types** section of our Drupal Administrator, and then click on the link that says **Add Content Type**. Following the steps from previous chapters, we can now create a Video content type by providing the following information:

- **Name**: **Video**
- **Type**: **video**
- **Description**: **Adds a new video to this website**

Once we have provided this information, we can now save this by clicking on the **Save content type** button at the bottom of the page. After the Video content type has been created, we can add our field that we will use to upload our videos to our system.

Adding a video file field

To add our file field to the Video content type, we will start by clicking on the link next to our newly created Video content type that says **manage fields**. Once the **manage fields** page opens up, we will then want to add a new file field to our video node type by providing the following information in the **Add** section:



Add

New field

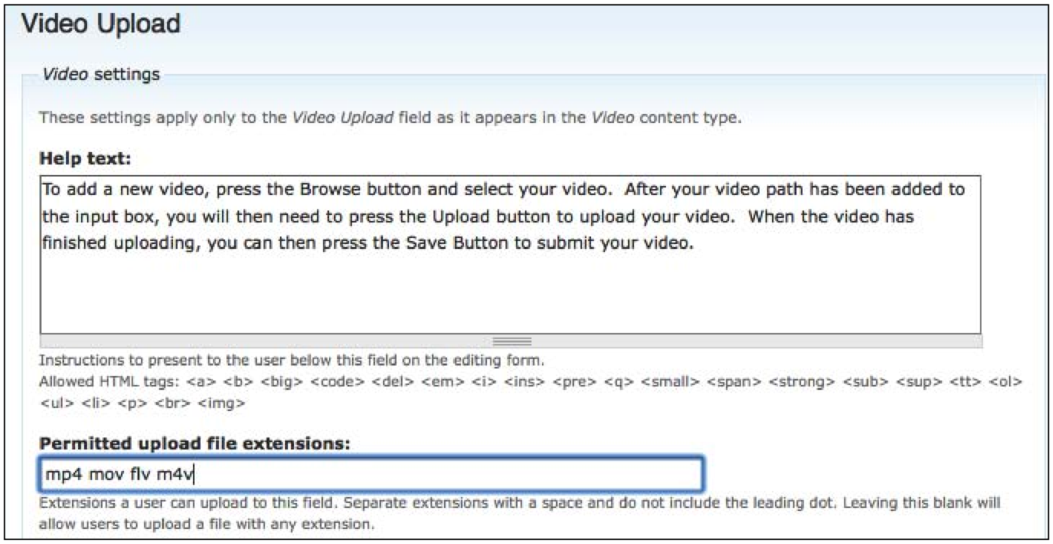
Video Upload
Label

field_ video
Field name (a-z, 0-9, _)

File
Type of data to store.

File Upload
Form element to edit the data.

After you click on the **Save** button at the bottom of the page, you will be given a new page where you can configure our video field. There are several things that we need to do on this page. The first thing we need to do is provide a very good help description that explains the steps involved in attaching a new video to the node. After this, we need to provide the permitted file extensions. This will filter any unwanted files, except for those that can be played as video in Flash.



Video Upload

Video settings

These settings apply only to the *Video Upload* field as it appears in the *Video* content type.

Help text:

To add a new video, press the Browse button and select your video. After your video path has been added to the input box, you will then need to press the Upload button to upload your video. When the video has finished uploading, you can then press the Save Button to submit your video.

Instructions to present to the user below this field on the editing form.
Allowed HTML tags: <a> <b> <big> <code> <del> <em> <i> <ins> <pre> <q> <small> <span> <strong> <sub> <sup> <tt> <ol> <ul> <li> <p>
 <img>

Permitted upload file extensions:

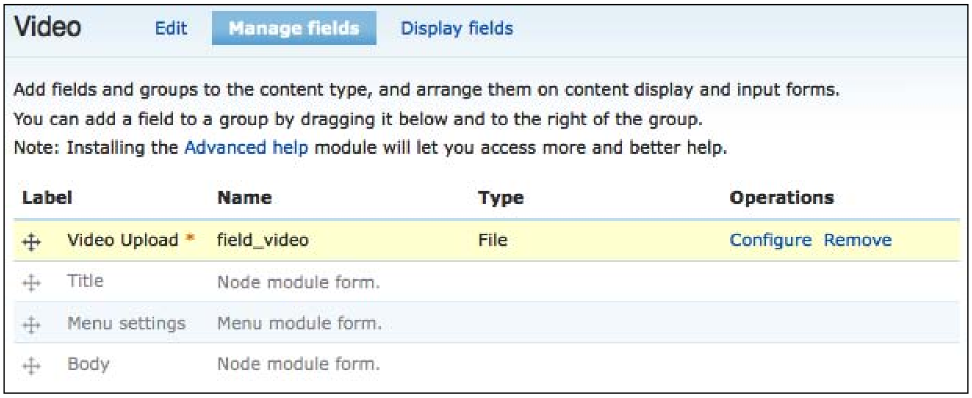
mp4 mov flv m4v

Extensions a user can upload to this field. Separate extensions with a space and do not include the leading dot. Leaving this blank will allow users to upload a file with any extension.

> The important thing to note here is the extensions allowed for our video upload. The reason we are limited to **mp4**, **mov**, **flv**, and **m4v** is because these are the video files that are compatible with Flash Player. If we wish to allow our web site to accept any video type, then that will require us to have some backend conversion utility such as FFmpeg. If this is a requirement for your web site, then I would highly recommend using either the FlashVideo or the Media Mover modules, which have support for backend video conversions using FFmpeg.

After we enter the help text and the file formats, we need to make sure that this field is required, since we do not want anyone submitting a video node without a video. We can do this by checking the checkbox that says **Required**. Now that we are done, we can click on the **Save field settings** button at the bottom of the page to save our new field.

After the field has been added to our video node type, we will set this field as the top most field, since the primary piece of content for our video nodes will be the video. We can do this by clicking on the ✣ symbol and then dragging the **Video Upload** field to the top of the list. After we do this, we need to make sure that we click on the **Save** button at the bottom of the page to commit the change.



Add fields and groups to the content type, and arrange them on content display and input forms.
You can add a field to a group by dragging it below and to the right of the group.
Note: Installing the Advanced help module will let you access more and better help.

Label	Name	Type	Operations
Video Upload *	field_video	File	Configure Remove
Title	Node module form.		
Menu settings	Menu module form.		
Body	Node module form.		

We have our Video content type ready to add videos to our site. Our next job is to install and configure the jQuery Media module, which we will use to show the submitted video content using our Video content type.

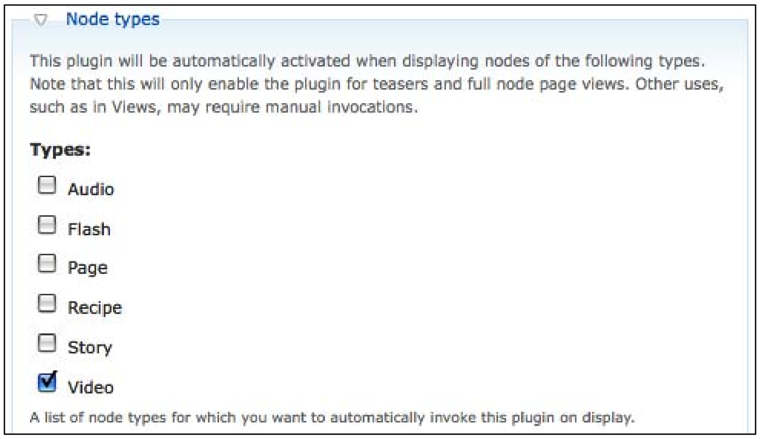
Installing and configuring the jQuery Media module

Installing the jQuery module can be done by simply downloading the jQuery Media module from `http://www.drupal.org/project/jquery_media`, and then placing the contents of this package in your site's `modules` folder. Now that we have this module in the right spot, we can enable it by navigating to the **Administrator | Modules** section of our Drupal site and clicking the checkbox next to **jQuery Media**, and then clicking on the **Save Configuration** button at the bottom of the page. Now that this module has been enabled, we will need to configure this module to show video on our web site.

Configuring the jQuery Media module

Configuring the jQuery Media module is a snap, and only requires us to visit one location within our Drupal administration. This location can be found by going to **Administer | jQuery Media**. This page will allow us to configure the jQuery Media module, so that we can view any video uploaded to our site.

To begin, we will need to enable this module for our Video content type, which we can do by simply expanding the **Node Types** section, and then checking the **Video** content type.



▽ Node types

This plugin will be automatically activated when displaying nodes of the following types. Note that this will only enable the plugin for teasers and full node page views. Other uses, such as in Views, may require manual invocations.

Types:

□ Audio

□ Flash

□ Page

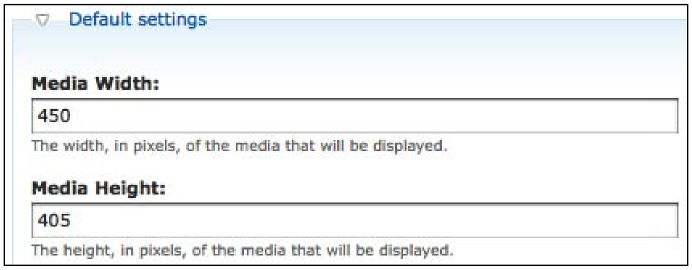
□ Recipe

□ Story

☑ Video

A list of node types for which you want to automatically invoke this plugin on display.

The next thing we will need to do is expand the **Default settings**, where we can provide a default width and height to our player.



▽ Default settings

Media Width:

450

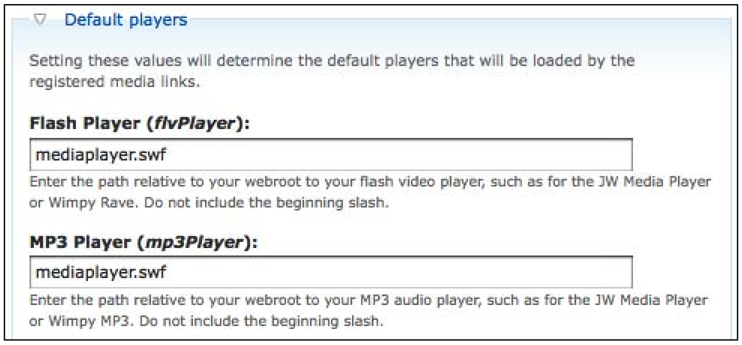
The width, in pixels, of the media that will be displayed.

Media Height:

405

The height, in pixels, of the media that will be displayed.

The last and final step will be to expand the **Default players** section, where we should then see the following:



▽ Default players

Setting these values will determine the default players that will be loaded by the registered media links.

Flash Player (*flvPlayer*):

mediaplayer.swf

Enter the path relative to your webroot to your flash video player, such as for the JW Media Player or Wimpy Rave. Do not include the beginning slash.

MP3 Player (*mp3Player*):

mediaplayer.swf

Enter the path relative to your webroot to your MP3 audio player, such as for the JW Media Player or Wimpy MP3. Do not include the beginning slash.

This section calls a Flash Player that will be used to play our media. Although we will be building our own video player, we can install a commercially available media player to view any videos that we submit.

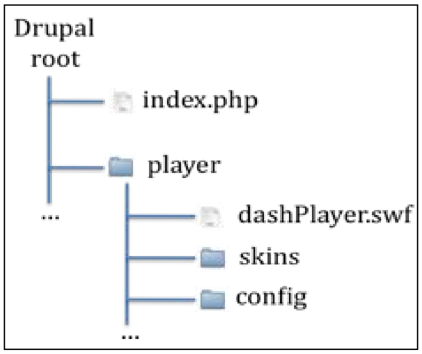
Installing a media player

There are several media players available to show video and audio content on our web site. Although each one of these players has their differences, all of them are fundamentally the same in how they play media on the site they are loaded. Following are the most popular media players that can be used with the jQuery Media module to show videos:

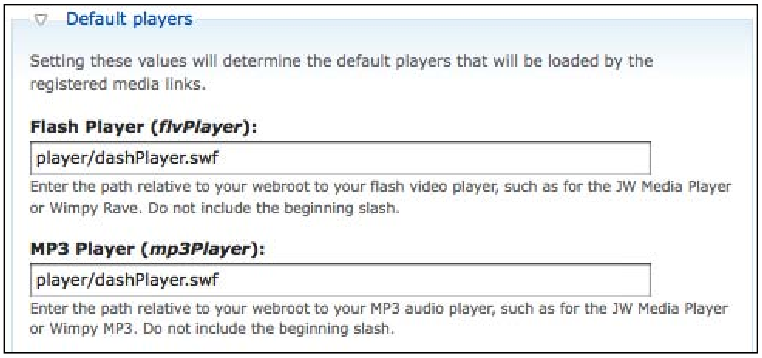
- Dash Media Player—`http://www.tmtdigital.com/project/dash_player`
- JW FLV Player—`http://www.longtailvideo.com/players/jw-flv-player/`
- Flow Player—`http://www.flowplayer.org`

All of these players are great, so I suggest you read up on all of them and pick whichever fits your needs. Regardless of which one you download, the following still applies.

Within each media player download, you should see a SWF file located at the root of the downloaded package. This file is used as the core player, and we need to remember the name of this file since we will use it to change the **Default player** for the jQuery Media module. Along with this file, there might be several additional files and directories that serve to complement the player's function, look, and feel. All we really need to do to install our player is create a folder at the root of our Drupal installation called `player`, and then place the contents of our downloaded player package inside that folder. For example, if we have downloaded the Dash Media Player, our Drupal folder structure should look like the following:



Now that we have our media player in the right spot, we will just need to change the **Default player** settings in the jQuery Media administrator to reflect this path to our media player.



▽ Default players

Setting these values will determine the default players that will be loaded by the registered media links.

Flash Player (*flvPlayer*):

player/dashPlayer.swf

Enter the path relative to your webroot to your flash video player, such as for the JW Media Player or Wimpy Rave. Do not include the beginning slash.

MP3 Player (*mp3Player*):

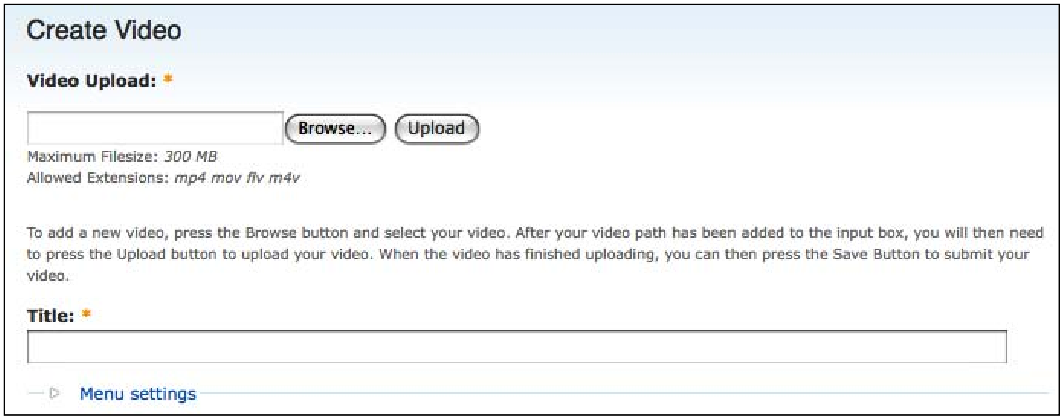
player/dashPlayer.swf

Enter the path relative to your webroot to your MP3 audio player, such as for the JW Media Player or Wimpy MP3. Do not include the beginning slash.

After we have done this, we can now click on the **Save Configuration** button at the bottom of the page to save our jQuery Media settings. Now that we have the jQuery Media module configured, our next step is to create a video node that we will use later to create our very own custom video player.

Creating a video node

To add a new video to our site, we will go to **Create Content | Video**. The first section that we will direct our attention to is the **Video Upload** field, which we created earlier in this chapter.



Create Video

Video Upload: *

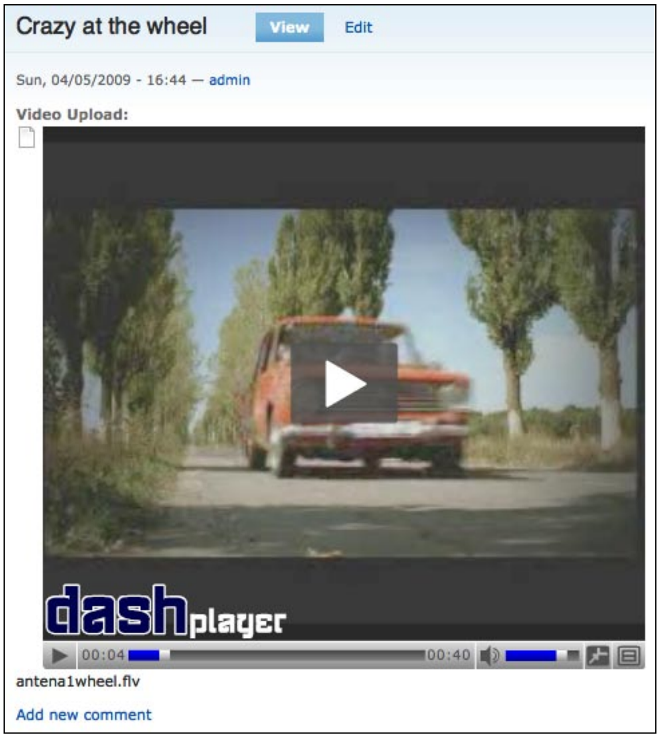
Maximum Filesize: *300 MB*
Allowed Extensions: *mp4 mov flv m4v*

To add a new video, press the Browse button and select your video. After your video path has been added to the input box, you will then need to press the Upload button to upload your video. When the video has finished uploading, you can then press the Save Button to submit your video.

Title: *

We can now add a new video file by clicking on the **Browse** button to search for a video file with an extension of MP4, FLV, MOV, or M4V. If you do not have any videos of this format, then you can easily download a sample video by going to `www.google.com` and typing "Sample FLV video file". Once you have your video on your local machine, and have selected it using the **Browse** functionality of the **Video Upload**, we can then attach that video to this node by clicking on the **Upload** button.

Once the video is done attaching itself to the node, we can then give our node a **Title**, and save our node by clicking on the **Save** button at the bottom of the page. Now, let's take a moment to give ourselves a pat on the back: we have successfully added video content to our Drupal web site.



Crazy at the wheel

Sun, 04/05/2009 - 16:44 — admin

Video Upload:

antena1wheel.flv

Add new comment

Our next task will be to build our very own custom video player to replace the commercial player used above.

Building a custom video player in Flash

For this section, we will now shift our focus to Flash, where we will build our very own custom video player to interface with Drupal. So, let's start by making a copy of the previous chapter's directory, and paste that copy as a new directory called `chapter6`. Once the files have been copied to the new directory, we will rename the `chapter5.fla` file to `chapter6.fla`, and then open up that project file along with the `main.as` file in Flash. Because the audio and video players share some similar functionalities, our first step will be to create a common class that can be used for both the audio and video functionalities.

Creating a MediaPlayer base class

Before we start creating a class to hold the functionality of video, we need to examine the contents of the `AudioPlayer.as` file that we created in the previous chapter. Looking at the code, we can see how some of the functionalities for audio can also be used for our video player functionality. As an example, in the `load` function of our `AudioPlayer.as` file, we can see how the video player would also implement the play and pause button functionality (since it can be played and paused), but the call to the sound object might not be shared between video and audio. The following highlighted code represents the code that can be shared between audio and video:

```
// Load an audio file.
public function load( file:String )
{
   // Setup the play button.
   playButton.buttonMode = true;
   playButton.mouseChildren = false;
   playButton.addEventListener(MouseEvent.MOUSE_UP,onPlay);
   // Setup the pause button.
   pauseButton.buttonMode = true;
   pauseButton.mouseChildren = false;
   pauseButton.addEventListener(MouseEvent.MOUSE_UP,onPause);
   // Set the state of the play and pause buttons.
   // We want to show the play button at first, so...
   playButton.visible = true;
   pauseButton.visible = false;
   // Load our sound file.
   sound.load( new URLRequest( file ) );
}
```

Whenever this situation arises, it is always best to create a new base class that will hold all the common functionalities between video and audio, and then have the VideoPlayer and AudioPlayer classes derive from that common class. This is the core concept behind **object-oriented programming** (**OOP**), which will allow our code to be easily maintained, reused, and expanded with additional functionality.

Since we have already created an `AudioPlayer` class in the previous chapter, we can create our common class by simply copying the file `AudioPlayer.as`, then creating a new file called `MediaPlayer.as`, and finally opening up that file in Flash.

Our first task will be to rename the class and constructor for this class to `MediaPlayer`, since that will be the name of our new base class.

```
// Declare our class
public class MediaPlayer extends MovieClip
{
   // Constructor function.
   // Called when someone creates a new MediaPlayer
   public function MediaPlayer()
   {
```

Another thing to note is that since our `MediaPlayer` class is using the `playButton` and `pauseButton` movie clips, we will need to add those variables to this file so that Flash will not throw an error when it cannot make the association to those movie clips.

Adding play and pause button instances to MediaPlayer

As mentioned before, the reason we need to take this step is because there are variables within our base class that represent objects within a subclass. Whenever this situation crops up, it is important to define the variables within the base class, so that Flash knows how to make that association to a subclass' child movie clips. This step requires us to not only add the variables to the `MediaPlayer` class, but also change a setting in the **Publish Settings** of our Flash project, so that Flash understands exactly what we are trying to do.

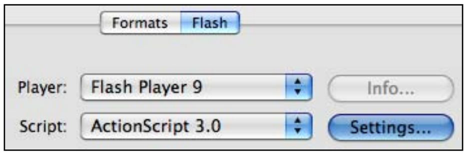
The first task here is to add the variables `playButton` and `pauseButton` to the `MediaPlayer`. This can be done pretty easily by adding the following variables to the bottom of our `MediaPlayer` class:

```
// Add the play and pause button to the media player.
public var playButton:MovieClip;
public var pauseButton:MovieClip;

// Declare our sound variable.
```

```
    private var sound:Sound;

    // Declare our sound channel
    private var channel:SoundChannel;

    // Variable to keep track of the audio position.
    private var position:Number;
```

After we add these variables, we need to make a change to our Flash project so that it reflects this change. Now, shifting our focus back to the `chapter6.fla` project file, we change the **Publish Settings** by navigating to **File | Publish Settings** in the Flash top menu.

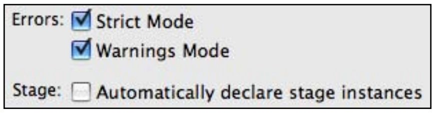
We make sure that we are in the Flash section of the settings by clicking on the **Flash** link in the settings bar. Once we are there, we click on the **Settings** button next to the **ActionScript 3.0** to enter the ActionScript settings for this project.



Player: Flash Player 9

Script: ActionScript 3.0

Now that we are in the ActionScript settings for this project, we make sure that we uncheck the **Automatically declare stage instances** checkbox.



Errors: ☑ Strict Mode
☑ Warnings Mode

Stage: ☐ Automatically declare stage instances

What we are doing here is telling ActionScript to not declare the movie clip instances, `playButton` and `pauseButton`, automatically in our classes, but that we will declare them ourselves (which we just did in the MediaPlayer class).

Now that we are done setting up the play and pause buttons for the MediaPlayer, we are ready to remove the rest of the uncommon code from the MediaPlayer class.

Removing uncommon code from MediaPlayer

Our final task will then be to go through the `MediaPlayer.as` file and remove any functionality that would not be common between video and audio. This will basically encompass any functionality that has to do with audio. One thing to constantly ask yourself when you are going through this process is, "Is this functionality something that both audio and video player would share?" If your answer to that question is no, then delete that functionality. When all is said and done, your base `MediaPlayer` class should look similar to the following:

```
package
{
   // Import all dependencies
   import flash.display.MovieClip;
   import flash.events.MouseEvent;
   // Declare our class
   public class MediaPlayer extends MovieClip
   {
      // Constructor function.
      // Called when someone creates a new MediaPlayer
      public function MediaPlayer()
      {
         // Make sure we call the MovieClip constructor
         super();
         // Let us know that we created this player.
         trace( "MediaPlayer created!" );
      }
      // Play a media file
      public function playFile()
      {
         // Show only the pause button.
         playButton.visible = false;
         pauseButton.visible = true;
      }
      // Pause a media file
      public function pause()
      {
         // Show only the pause button.
         playButton.visible = true;
         pauseButton.visible = false;
      }
      // Load a media file.
      public function load( file:String )
      {
         // Setup the play button.
         playButton.buttonMode = true;
         playButton.mouseChildren = false;
         playButton.addEventListener(MouseEvent.MOUSE_UP,onPlay);
         // Setup the pause button.
         pauseButton.buttonMode = true;
         pauseButton.mouseChildren = false;
         pauseButton.addEventListener(MouseEvent.MOUSE_UP,onPause);
         // Set the state of the play and pause buttons.
         // Set the state of the play and pause buttons.
         // We want to show the play button at first, so...
         playButton.visible = true;
         pauseButton.visible = false;
      }
      // Called when the play button has been pressed.
```

```
        private function onPlay( event:MouseEvent )
        {
           // Play the audio track.
           playFile();
        }
        // Called when the user presses the pause button.
        private function onPause( event:MouseEvent )
        {
           // Pause the audio track.
           pause();
        }
        // Add the play and pause button to the media player.
        public var playButton:MovieClip;
        public var pauseButton:MovieClip;
     }
  }
```

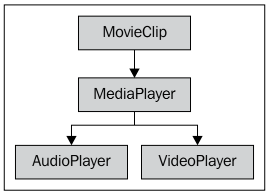
Now that we have our common `MediaPlayer` class, we can modify our `AudioPlayer.as` file, so that it derives this common functionality from the `MediaPlayer` class.

Modifying the AudioPlayer class to derive from MediaPlayer

To modify the `AudioPlayer` class, we will first need to open up the `AudioPlayer.as` file that contains the original code from our previous chapter. The first thing we need to do is make the AudioPlayer class derive from the MediaPlayer class that we just created. This can be done by adding the `MediaPlayer` class after the `extends` keyword:

```
// Declare our class
public class AudioPlayer extends MediaPlayer
```

By doing this, we are basically saying that the `AudioPlayer` class is inheriting functionality from the `MediaPlayer` class, and we will do the same when we create the `VideoPlayer` class. This creates a hierarchy of functionality most commonly referred to as inheritance in object-oriented programming. Shown in a graphical representation of the class structure, we can now represent the **AudioPlayer**, **VideoPlayer**, and **MediaPlayer** classes as the following:



There is some terminology that is important for us to remember when working with this hierarchical structure of class functionality. In ActionScript 3.0, the base class is referred to as super class, and can be referenced in any of the subclasses by using the keyword `super`. This is important to note since we will be using this keyword in the following section, where we extend the functionality of the super class `MediaPlayer` within the subclass `AudioPlayer`.

Since we have already inherited functionality from the `MediaPlayer` class, our next task will be to walk through the `AudioPlayer` class and remove any functionality that is provided from the `MediaPlayer` class. Our task also includes overriding certain functions whose functionality will be extended with the `AudioPlayer` class.

Extending and overriding base (super) class functionality

Our first task here will be to remove all the `MediaPlayer` class functionalities from the `AudioPlayer` class. We can do this by simply deleting the play and pause button functionality that we provided within the `MediaPlayer` class. After we have done this, our `AudioPlayer.as` file should look as follows:

```
package
{
   // Import all dependencies
   import flash.media.Sound;
   import flash.media.SoundChannel;
   import flash.net.URLRequest;
    // Declare our class
   public class AudioPlayer extends MediaPlayer
   {
      // Constructor function.
      // Called when someone creates a new AudioPlayer
      public function AudioPlayer()
      {
         // Make sure we call the MediaPlayer constructor
         super();
         // Make sure to create our sound object
         sound = new Sound();
         // Initialize the position.
         position = 0;
         // Let us know that we created this player.
         trace( "AudioPlayer created!" );
      }
      // Play an audio file
         override public function playFile()
      {
         // Play our sound file.
         channel = sound.play(position);
      }
      // Pause an audio file
      override public function pause()
      {
         // Save the channel position.
         position = channel.position;
         // Stop our sound file.
         channel.stop();
      }
      // Load an audio file.
      public function load( file:String )
      {
         // Load our sound file.
         sound.load( new URLRequest( file ) );
      }
       // Declare our sound variable.
      private var sound:Sound;
       // Declare our sound channel
      private var channel:SoundChannel;
      // Variable to keep track of the audio position.
      private var position:Number;
   }
}
```

After we have done that, we will need to override certain functions where the AudioPlayer extends the functionality of the MediaPlayer. In ActionScript 3.0, we can override any public or protected functions from the base class by using the keyword override when declaring the function. For example, in the MediaPlayer class, there is a function called playFile, which we will need to override in our AudioPlayer class to actually play the file. But, since we want to keep the functionality of the MediaPlayer class, which sets the play and pause button states, we also want to make sure that we call the super.playFile in our overridden function to make sure that we get the full functionality. By calling super.playFile, we are telling ActionScript to call the playFile function of the super class MediaPlayer. By following these rules, our playFile function in the AudioPlayer class should look like the following:

```
// Play an audio file
override public function playFile()
{
   // Call the MediaPlayer playFile function.
   super.playFile();
   // Play our sound file.
   channel = sound.play(position);
}
```

It is also important to note that if we ever wish to completely override a function from a super class, we would leave out the super function call (`super.playFile`). But since we wish to extend functionality, as opposed to override functionality, we will leave the super function call so that our super class `MediaPlayer` has a chance to add its functionality to the `playFile` function.

Taking this into account, we now have enough knowledge to modify the pause and load functions within our `AudioPlayer` class, so that it correctly extends functionality from the `MediaPlayer` class.

```
// Play an audio file
override public function playFile()
{
   // Call the MediaPlayer playFile function.
   super.playFile();
   // Play our sound file.
   channel = sound.play(position);
}
   // Pause an audio file
   override public function pause()
{
   // Call the MediaPlayer pause function.
   super.pause();
   // Save the channel position.
   position = channel.position;
   // Stop our sound file.
   channel.stop();
}
   // Load an audio file.
override public function load( file:String )
{
   // Call the MediaPlayer load function.
   super.load( file );
   // Load our sound file.
   sound.load( new URLRequest( file ) );
}
```

We are now done modifying our `AudioPlayer` class to extend the `MediaPlayer` functionality. Before we move onto the next section, it is very important for us to run this code to make sure that it still functions as it did before we made our change to class hierarchy. If all is well, we can move on to creating a new `VideoPlayer` class.

Creating a VideoPlayer class

The first step we will take to create a video player class is copy the `AudioPlayer.as` file to a new file called `VideoPlayer.as`. After we open up this new file in Flash, we can create a stub class, where we will remove any of the audio functionality (leaving only the super class functionality). After removing all audio functionality, our `VideoPlayer` class should look like the following:

```
package
{
   // Declare our class
   public class VideoPlayer extends MediaPlayer
   {
      // Constructor function.
      // Called when someone creates a new VideoPlayer
      public function VideoPlayer()
      {
      // Make sure we call the MediaPlayer constructor
         super();
         // Let us know that we created this player.
         trace( "VideoPlayer created!" );
      }
      // Play a video file
      override public function playFile()
      {
         // Call the MediaPlayer playFile function.
         super.playFile();
      }
      // Pause a video file
      override public function pause()
      {
         // Call the MediaPlayer pause function.
         super.pause();
      }
      // Load a video file.
      override public function load( file:String )
      {
         // Call the MediaPlayer load function.
         super.load( file );
      }
   }
}
```

Our next task will be to add the Flash video functionality that utilizes these functions to play and pause video. There are three different classes that are used to show and manipulate video in Flash, which are `Video`, `NetStream`, and `NetConnection`.

Working with Video, NetStream, and NetConnection

The first thing that we will need to do is make sure that we import the functionality from these three classes as well as any dependent classes used by these classes. We can do that by including the following code at the top of our VideoPlayer class as follows:

```
package
{
   // Import all dependencies
   import flash.media.Video;
   import flash.net.NetStream;
   import flash.net.NetConnection;
   import flash.net.ObjectEncoding;
   import flash.events.*;

   // Declare our class
   public class VideoPlayer extends MediaPlayer
   {
```

After we import those classes, our next task will be to create variables within this class that we will use to add video functionality. We can do this pretty easily as follows:

```
// Load a video file.
override public function load( file:String )
{
   // Call the MediaPlayer load function.
   super.load( file );
}
// Add all of our video variables.
private var video:Video;
private var stream:NetStream;
private var connection:NetConnection;
```

After these variables have been added to your VideoPlayer class, we can now add the video functionality that will show a video stream in our custom media player. We will start with initializing all of our video variables.

Initializing our video variables

Our first task when adding video functionality is to make sure that all the variables have been initialized to stream video. We can do this by adding some new functions to our `VideoPlayer` class called `connect` and `setupVideoStream`, which we will use to set up the `connection` and `stream` variables respectively. We can then call those functions from within the constructor of our `VideoPlayer`. When initializing the `connection` and `stream` variables, we will also need to create `onError` and `onStatus` callback functions, to be called when the status changes or an error occurs. Another thing to note when looking at this code is that we will need to include an empty function called `onMetaData`. This is simply a stub function to keep Flash from throwing errors when the client association tries to call this function. The additions to our `VideoPlayer` class should look like the following:

```
// Constructor function.
// Called when someone creates a new VideoPlayer
public function VideoPlayer()
{
   // Make sure we call the MediaPlayer constructor
   super();
   // Let us know that we created this player.
   trace( "VideoPlayer created!" );
   // Connect to our NetConnection.
   connect();
   // Setup the video stream.
   setupVideoStream();
}
// Create a new NetConnection
private function connect()
{
   connection = new NetConnection();
   connection.addEventListener(NetStatusEvent.NET_STATUS,onStatus);
   connection.addEventListener(SecurityErrorEvent.SECURITY_ERROR,
   onError);
   connection.objectEncoding = ObjectEncoding.AMF0;
   connection.connect(null);
}
// Setup a new video stream.
private function setupVideoStream()
{
   stream = new NetStream(connection);
   stream.addEventListener(NetStatusEvent.NET_STATUS,onStatus);
   stream.addEventListener(AsyncErrorEvent.ASYNC_ERROR, onError);
   stream.client = this;
}
```

```
// Stub function for the client association.
public function onMetaData(info:Object) {}

// Our video status handler.
private function onStatus(event:NetStatusEvent)
{
}

// Called when a video error occurs.
private function onError(event:Object):void
{
   trace("VideoPlayer Error: " + event);
}
```

Our final task in initializing all the variables is to create the video object that we will use to show our video on the screen.

Creating the video object

In order to create the video object, we must first instantiate it with the width and the height of the video area we wish to show. This is where it might get a little fuzzy since we really do not want to hard code a width and height for our video. A better solution is to design our software so that the size of the embedded player determines the size of the video. This can be done by referencing the `stage` object within our `VideoPlayer` class, and then setting the width and height of our video to the same size.

This, however, presents an unusual caveat since our `VideoPlayer` must first be added to the stage in order to make the `stage` variable valid. Fortunately, we can design our class to trigger an event when our class is added to the stage, and then set the width and height of our video to the stage width and height within the handler function of this event. Once we have the size of our video player, we can assign the `NetStream` to our video object using the `attachNetStream` function.

Once we have declared our new video object, we will need to add it to the `VideoPlayer` so that it is visible to our users. In order to do this properly, we will need to place it behind the play and pause buttons, so that it will not cover them up when the video is playing. This can be done using the `addChildAt` function, and then providing an index of 0, which means to place it behind every object within the `VideoPlayer` class.

```
// Constructor function.
// Called when someone creates a new VideoPlayer
public function VideoPlayer()
{
   // Make sure we call the MediaPlayer constructor
   super();
```

```
      // Let us know that we created this player.
      trace( "VideoPlayer created!" );
      // Connect to our NetConnection.
      connect();
      // Setup the video stream.
      setupVideoStream();
      // Add a listener when the player is added to the stage.
      addEventListener( Event.ADDED_TO_STAGE, onAdded );
   }
   // Called when the video player has been added to the stage.
   private function onAdded( event:Event )
   {
      // Create our video object the size of our stage.
      video = new Video( stage.width, stage.height );
      // Attach our net stream to the video object.
      video.attachNetStream( stream );
      // Add the video to the VideoPlayer.
      addChildAt( video, 0 );
   }
```

We are now done initializing all the variables for our `VideoPlayer` class. Our next task will be to hook up the functionality.

Adding video functionality

To add video functionality, we will now shift our focus to the load, play, and pause functions, where we will play with our video stream to perform this functionality. We will start with the load functionality.

Adding video load

For the video load function, we will want to load the video file passed to the function but not play it. Unfortunately, the video stream class does not support a load function, but we can simulate it by playing the file and then pausing it once it starts to play. This will trigger our `VideoPlayer` class to start loading the video file. To do this, we will need to utilize two different functions where we play the video stream, and then handle the play status in the `onStatus` function, where we will pause the stream. In order to make this work correctly, we will need to create a class variable that will keep track if the video file has been loaded, and only pause the video if it has not been loaded. This functionality looks like the following:

```
   // Constructor function.
   // Called when someone creates a new VideoPlayer
   // Constructor function.
   // Called when someone creates a new VideoPlayer
   public function VideoPlayer()
```

```
{
   // Make sure we call the MediaPlayer constructor
   super();
   // Let us know that we created this player.
   trace( "VideoPlayer created!" );
   // Connect to our NetConnection.
   connect();
   // Setup the video stream.
   setupVideoStream();
   // Initialize to not loaded.
   loaded = false;
   // Add a listener when the player is added to the stage.
   addEventListener( Event.ADDED_TO_STAGE, onAdded );
}
// Our video status handler.
private function onStatus(event:NetStatusEvent)
{
   // If the video is playing.
   if( event.info.code == "NetStream.Play.Start" )
   {
      // Pause the stream if it is not loaded.
      if( !loaded ) {
         loaded = true;
         stream.pause();
      }
   }
}
// Load an audio file.
override public function load( file:String )
{
// Call the MediaPlayer load function.
   super.load( file );
   // Reset the loaded flag.
   loaded = false;
   // stop the current stream.
   stream.close();
   // Start playing the new stream.
   stream.play(file);
}
// Add all of our video variables.
private var video:Video;
private var stream:NetStream;
private var connection:NetConnection;
// Variable to keep track of loaded state.
private var loaded:Boolean;
```

Now that we have load functionality, our next task is to create the play and pause functionality.

Adding play and pause functionality

Fortunately, adding the play and pause functionality to the `VideoPlayer` class is very simple, and requires only a single function call using the stream variable like the following:

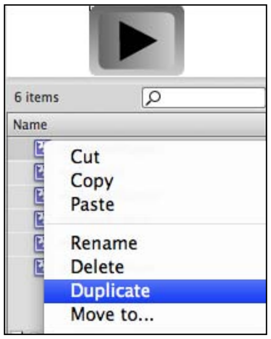
```
// Play a video file.
override public function playFile()
{
   // Call the MediaPlayer playfile function.
   super.playFile();
   // Resume the stream.
   stream.resume();
}
// Pause a video file
override public function pause()
{
   // Call the MediaPlayer pause function.
   super.pause();
    // Pause the stream.
      stream.pause();
}
```

Now that we are done adding the play and pause functionality, we can shift our focus to our `chapter6.fla` file, where we will utilize this new class to show video.

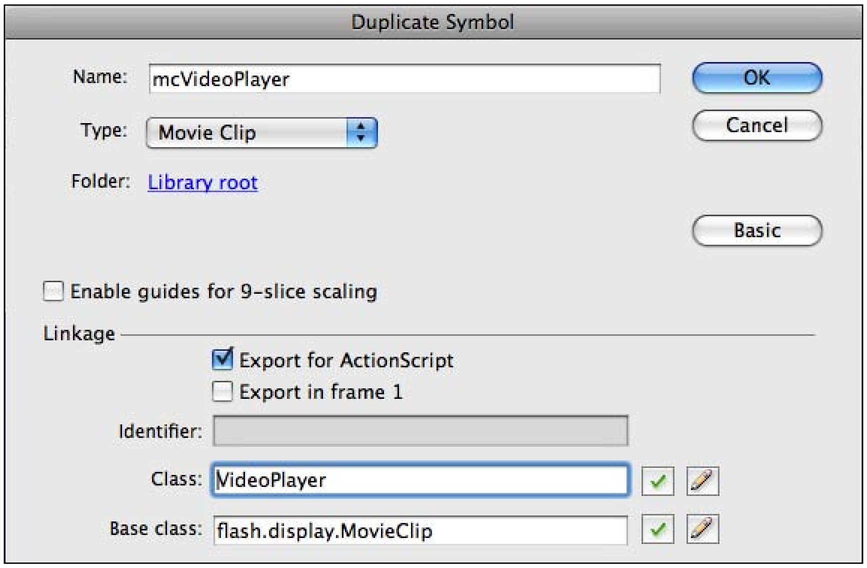
Creating a new VideoPlayer MovieClip

We can start out this section by opening up the `chapter6.fla` file, where we will then direct our attention to the Library section. This is where we will set up our new custom video player.

The first thing that we would like to do is create a duplicate of the `mcAudioPlayer` movie clip, and then change the properties so that it has a different functionality. To do this, we will first right-click on the **mcAudioPlayer** movie clip and select the option **Duplicate**.



This will then bring up a new window, where we can create a new movie clip for our video player. We will also need to make sure to check the **Export for ActionScript** and then provide **VideoPlayer** for the class.



Duplicate Symbol

Name: mcVideoPlayer

Type: Movie Clip

Folder: Library root

□ Enable guides for 9-slice scaling

Linkage

☑ Export for ActionScript

□ Export in frame 1

Identifier:

Class: VideoPlayer

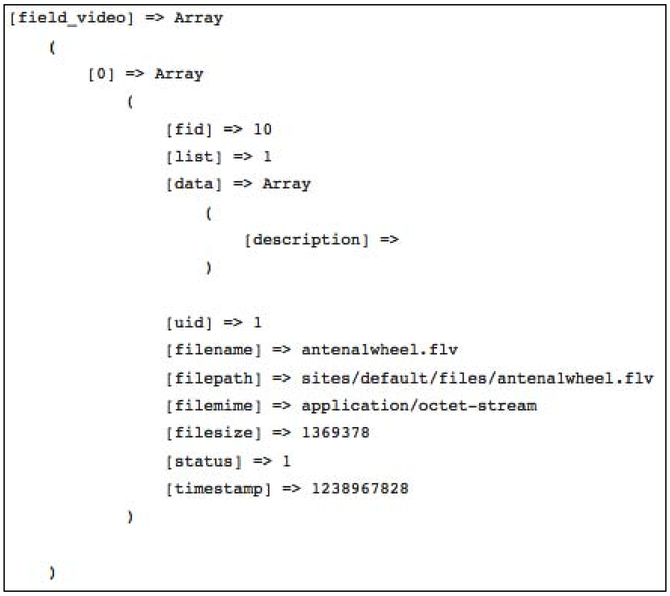
Base class: flash.display.MovieClip

As mentioned in the previous chapter, this will create a link between the movie clip `mcVideoPlayer` and the class `VideoPlayer` that we just created. Now that we have created our **mcVideoPlayer** movie clip, we need to navigate back to the root of our Flash project and then delete the **mcAudioPlayer** instance from the stage. This is done so that we can programmatically select which player to use depending on the media within our Drupal node. When we are done with this step, we should be left with only the Title region of our Flash application.

We are now ready to modify the `onNodeLoad` function within our `main.as` file to use our new **VideoPlayer**, and to reference the video attached to our Drupal node, but first, we need to take a look at the Services Administrator to determine the node structure of a video node.

Linking the VideoPlayer to Drupal

Shifting our focus back to Drupal, we will first determine the node structure of the video node that we created at the beginning of this chapter. Again, the Services Administrator comes to our rescue, where we can use the `node.get` routine to examine the video node that we created earlier in this chapter. Once we enter the node ID in the **nid** input box and click on the **Call Method** button, we should see the following results in the Results section. We can then examine the result, where we will notice the **field_video** FileField that we created.



```
[field_video] => Array
    (
        [0] => Array
            (
                [fid] => 10
                [list] => 1
                [data] => Array
                    (
                        [description] =>
                    )

                [uid] => 1
                [filename] => antenalwheel.flv
                [filepath] => sites/default/files/antenalwheel.flv
                [filemime] => application/octet-stream
                [filesize] => 1369378
                [status] => 1
                [timestamp] => 1238967828
            )

    )
```

Looking at this information, we can determine that the node data that we are interested in can be represented by the following code, given the node object:

```
node.field_video[0]["filepath"]
```

We can now take that information back to our `main.as` file, where we can modify the `onNodeLoad` function to load this video.

Loading and playing our Drupal video

Moving our focus back to the `main.as`, we can now examine the `onNodeLoad` function, which up to this point, should look like the following:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
   // Declare our base URL.
   var fileURL:String = baseURL;
   // Add our file's relative path.
   fileURL += "/";
   fileURL += node.audio.file.filepath;
   // Play our audio file
   player.load( fileURL );
}
```

Our first task will be to programmatically allow ourselves to either create an `AudioPlayer` or a `VideoPlayer`, depending on the type of media that is attached to our node. To start with, we will need to modify our existing code within the `onNodeLoad` function so that our `AudioPlayer` is only created when the audio is provided within the node. To do this, we will first declare a generic player variable, and then set the value of that player when we know for sure what kind of player it should be.

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
   // Declare our base URL.
   var fileURL:String = baseURL;
   // Add our file's relative path.
   fileURL += "/";
   // Declare a generic media player.
   var player = null;
   // If this node has audio.
   if( node.audio ) {
   // Declare our player as an AudioPlayer.
   fileURL += node.audio.file.filepath;
   player = new AudioPlayer();
   }
   // Play our audio file
   player.load( fileURL );
}
```

After we have done this, we can now do the same for the `VideoPlayer` if we find the `field_video` within the node object.

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
   // Declare our base URL.
   var fileURL:String = baseURL;
   // Add our file's relative path.
   fileURL += "/";
   // Declare a generic media player.
   var player = null;
   // If this node has audio.
   if( node.audio ) {
      // Declare our player as an AudioPlayer.
      fileURL += node.audio.file.filepath;
      player = new AudioPlayer();
   }
   else if( node.field_video ) {
      // Declare our player as a VideoPlayer.
      fileURL += node.field_video[0]["filepath"];
      player = new VideoPlayer();
   }
   // Play our audio file
   player.load( fileURL );
}
```

Finally, we need to make sure to add our player as a child to stage. We can do this by using the `addChild` function, which will trigger the `onAdded` function to trigger within our VideoPlayer class.

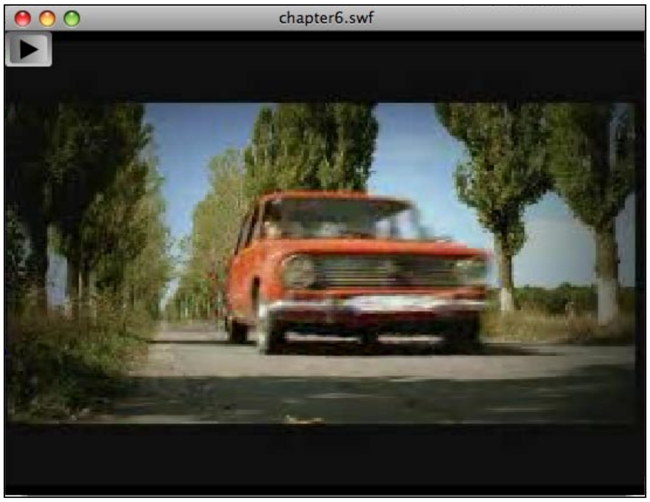
```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
   // Declare our base URL.
   var fileURL:String = baseURL;
   // Add our file's relative path.
   fileURL += "/";
   // Declare a generic media player.
   var player = null;
   // If this node has audio.
```

```
      if( node.audio ) {
         // Declare our player as an AudioPlayer.
         fileURL += node.audio.file.filepath;
         player = new AudioPlayer();
      }
      else if( node.field_video ) {
         // Declare our player as a VideoPlayer.
         fileURL += node.field_video[0]["filepath"];
         player = new VideoPlayer();
      }
      // Add the player to the stage.
      addChild( player );
      // Play our audio file
      player.load( fileURL );
   }
```

We can now temporarily provide the `nodeID` for our video node at the top of the `main.as` file so that we can try out our new video player.

```
   // Declare our variables
   var baseURL:String = "http://localhost/drupal6";
   var gateway:String = baseURL + "/services/amfphp"
   var sessionId:String = "";
   var nodeId:Number = 9;
```

Now, when we run our project, we should see a working video with play and pause functionality!



We are now ready to take this application back to Drupal.

Adding our custom media player to Drupal

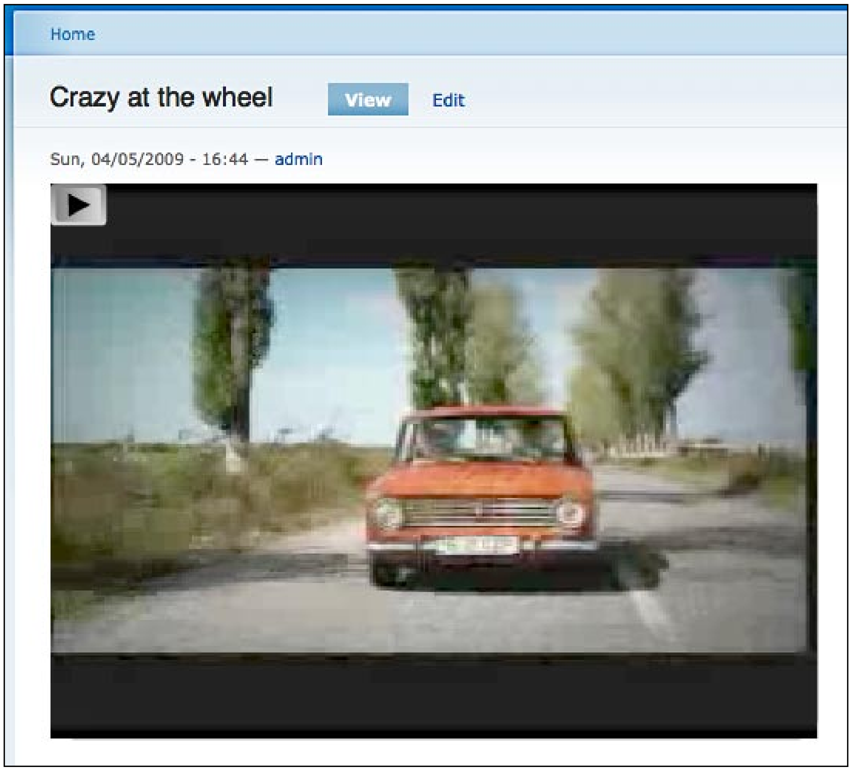
The first step we will need to take is to change our `nodeId` variable back to the FlashVars format so that we can compile our new media player for Drupal.

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp"
var sessionId:String = "";
var nodeId:Number = root.loaderInfo.parameters.node;
```

After we run our media player with these changes (which compiles the SWF file), we can follow the steps from Chapter 4 and upload our new media player by navigating to **Create Content | Flash**. Once we have our Flash application submitted to Drupal, we can create a template for both the audio and video nodes using the **Content Templates** within the Drupal Administrator. Assuming that the node ID for our flash node application is 11, our body templates should look like the following for both the video and audio nodes:

```
<?php
// Get the Flash application.
$sql = "SELECT * FROM {node} WHERE nid=%d";
$flash = db_fetch_object(db_query($sql, 11));
// Load the flash node.
flashnode_load( $flash );
// Set the flashvars to the right node Id.
$flash->flashnode["flashvars"] = 'node=' . $node->nid;
// Show the Flash application.
print theme('flashnode', $flash->flashnode, FALSE);
?>
```

Once we have our templates in place, we can navigate to any audio and video node within our Drupal website and see our new media player in action!



Home

Crazy at the wheel

Sun, 04/05/2009 - 16:44 — admin

Summary

This chapter covered a lot of material. The following is a list of the key topics covered:

- We started out with a brief overview of four different video solutions for Drupal, and decided to implement the FileField + jQuery Media for our use case
- We successfully set up a working video solution for our Drupal web site by combining a commercially available media player with the FileField and jQuery Media module
- We created a base class called `MediaPlayer` to hold the common functionality between video and audio

- We discussed some key object-oriented features in ActionScript 3.0 such as extends, override, and super keywords as well as the meaning of inheritance
- We redesigned our `AudioPlayer` class to use our new object-oriented structure
- We built a `VideoPlayer` class that utilizes Flash classes to display video
- We created some functionality to dynamically select which media player to declare, depending on the type of media attached to our Drupal node
- We added our new media player to Drupal

In the next chapter we will discuss how external components can interact with our Flash applications in Drupal (including other Flash components) by building a remote control for our new custom media player.

7
The Hybrid Approach
Part 1: Componentization

When integrating Flash with Drupal, one common obstacle that most people face is creating architecture that delivers the Flash content while, at the same time, utilizes the advantages of Drupal content management. This creates a problem because Drupal was designed and implemented with an HTML/JavaScript user interface, which tends to have a tightly woven bond to the server-side business logic that controls the function of the site. Because of this, many Flash implementations for a Drupal web site tend to have an all-or-nothing approach when integrating Flash as the user interface. What many people do not realize is that another approach exists. In this approach, the Flash integration takes the form of a collection of widgets, each with their own specific functions that are able to communicate to one another as if they were built into a single Flash application. In my opinion, this *hybrid* approach is ideal because it allows us to pick and choose the components that we wish to have as Flash on our Drupal site and the ones that we would like to keep as HTML, giving us the best of both worlds.

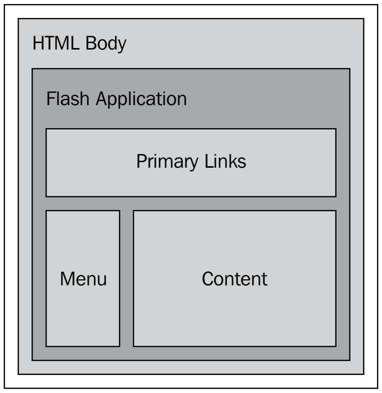
In the next two chapters we will discuss how to build a Drupal site where the Flash integration is componentized. We will also discuss how each of these Flash widgets can communicate with one another as if they were combined into one application by creating a *Remote Control* for the media player we built in the previous chapter. This example will easily illustrate the power of this technique when developing a Flash-integrated Drupal web site. We will cover the following topics in this chapter:

- What is the hybrid approach?
- Creating a media player control bar
- Creating a communication gateway
- Static functions and the this pointer
- Adding the `ControlBar` to our Flash project

What is the hybrid approach?

Many Flash web sites today are constructed as a single Flash application (SWF) that is built to behave like a web site. Although this approach is great for many use cases, it does not work well with content management systems. This is because the content in content management systems is constantly changing, or there are many different forms of content such as comments, user profiles, and so on. For these web sites, an approach is needed where Flash and HTML are inter-mixed to give the best user interface for each component on the web site. This approach is called a *hybrid* approach, where we integrate Flash into Drupal.

To illustrate this, let us take an example of a typical all-or-nothing Flash implementation for a Drupal web site, where the entire user interface is provided through Flash. In the following illustration we can see a block diagram showing how a Flash application would look if it was embedded to take up the entire page.

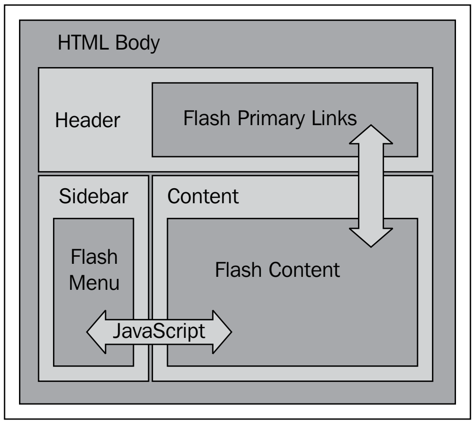


This diagram may look very familiar since it is typical for most Flash-driven Drupal web sites. However, there are several problems in using this type of architecture. These problems can be explained with the help of the following points:

1. Static layout—A static layout is one that does not resize when the browser viewing the page resizes. This is a very common issue with Flash web sites, where the web developer must pick a common browser size and then design the layout of the whole Flash application to fit that size. A fluid layout will automatically resize with the browser and is usually preferred for most use cases. Standard HTML allows fluid layouts.
2. All-or-nothing development—Integrating Flash into your Drupal web site is usually a very frustrating process because it has a tendency to spider web its way through your entire site, until it completely takes over. In most cases this makes the development time much longer than if the site were designed and deployed using only HTML.

3. Large and cumbersome application—When a single Flash application is used to deploy your entire user interface for Drupal, the end product is almost always large and cumbersome. This makes your bandwidth usage greater when your site is deployed to a large audience and it will slow down the speed of your site.
4. Not search engine friendly—Although much improvement has been made with Flash content being indexed with popular search engines, it still is no comparison to HTML-driven web content.

The hybrid approach, however, breaks apart this massive Flash application into separate components (or widgets), where each widget is in charge of a single piece of functionality, and then communicates to the other widgets using a JavaScript gateway. This approach is illustrated as follows:



By using this architecture we can use the facilities that Drupal provides when placing content on the page, such as using Blocks and Panels—which we will cover in the next chapter. It also forces us to break our Flash widgets into smaller applications, where each widget performs a specific task. In software development this is commonly referred to as componentization. Componentization makes our Flash implementation expandable and easier to maintain. Although this architecture may be more flexible, it does require a communication protocol between each widget on the page, which we will simulate by creating a remote control for our media player from the previous chapter. But first, we must abstract out all the functionality that we would like to use as the separate remote control application, namely the control bar.

Creating a media player control bar

Before we begin this section, we will first need to copy all the files used for Chapter 6 and create a new folder to hold all of our changes made for this chapter. Once we are done doing this, our first task will be to change the architecture of our current `MediaPlayer` so that it will support the separation between the media and control bar functionality. In order to do this, we will need to create an abstraction between the media and controls so that their functionality may be within the same Flash application or in two separate Flash applications connected together with a JavaScript gateway. By performing this abstraction we will understand the importance and benefits of object-oriented practices which allows this type of abstraction. To start, we will first extract all of the play and pause button functionality into a new class that we will call `ControlBar`. The main purpose of this class will be to hold all the user interaction for the media player.

Creating a ControlBar class

Let's start this section by copying the `MediaPlayer.as` file, and then creating a new file from that copy called `ControlBar.as` in the same directory. The reason we are copying the `MediaPlayer.as` file is simply because the `MediaPlayer` class currently holds the functionality that we will use for our new `ControlBar` class.

Once we have done this, we are ready to open up the `ControlBar.as` file and modify it so that it represents the correct class name, and change the constructor function given as follows. When we are done, our class definition should look something similar to the following, where the changes have been highlighted:

```
package
{
   // Import all dependencies
   import flash.display.MovieClip;
   import flash.events.MouseEvent;
   // Declare our class
   public class ControlBar extends MovieClip
   {
      // Constructor function.
      // Called when someone creates a new ControlBar
      public function ControlBar()
      {
         // Make sure we call the MovieClip constructor
         super();
         // Let us know that we created the ControlBar.
         trace( "ControlBar created!" );
      }
...
...
```

We will also need to move the playButton and pauseButton initialization into the constructor of this class, since we do not want any dependency on loaded media in order to instantiate our buttons. We can do this by first copying all the contents of the load function and moving that code into the constructor, and then completely deleting the load function. When we are done, our ControlBar class should look like the following:

```
package
{
   // Import all dependencies
   import flash.display.MovieClip;
   import flash.events.MouseEvent;
   // Declare our class
   public class ControlBar extends MovieClip
   {
      // Constructor function.
      // Called when someone creates a new ControlBar
      public function ControlBar()
      {
         // Make sure we call the MovieClip constructor
         super();
         // Let us know that we created the ControlBar.
         trace( "ControlBar created!" );
         // Setup the play button.
         playButton.buttonMode = true;
         playButton.mouseChildren = false;
         playButton.addEventListener(MouseEvent.MOUSE_UP,onPlay);
         // Setup the pause button.
         pauseButton.buttonMode = true;
         pauseButton.mouseChildren = false;
         pauseButton.addEventListener(MouseEvent.MOUSE_UP,onPause);
         // Set the state of the play and pause buttons.
         // We want to show the play button at first, so...
         playButton.visible = true;
         pauseButton.visible = false;
      }
      // Play a media file
      public function playFile()
      {
         // Show only the pause button.
         playButton.visible = false;
         pauseButton.visible = true;
      }
      // Pause a media file
      public function pause()
      {
         // Show only the pause button.
```

```
            playButton.visible = true;
            pauseButton.visible = false;
         }
         // Called when the play button has been pressed.
         private function onPlay( event:MouseEvent )
         {
            // Play the audio track.
            playFile();
         }
         // Called when the user presses the pause button.
         private function onPause( event:MouseEvent )
         {
            // Pause the audio track.
            pause();
         }
         // Add the play and pause button to the media player.
         public var playButton:MovieClip;
         public var pauseButton:MovieClip;
      }
   }
```

Our next task is to modify the `MediaPlayer.as` file so that it no longer has a dependency on the `ControlBar` class. Let's now open up the `MediaPlayer.as` file so that we can make all the appropriate changes.

Removing the ControlBar dependency from MediaPlayer

In this section we will start by opening up the `MediaPlayer.as` file and removing any dependencies that this file has on the `ControlBar` object. Although we will temporarily break this class, we will make the connection again when we build a communication gateway later in this chapter. For now, we simply want to remove any trace of a control bar until our `MediaPlayer.as` file looks like the following:

```
package
{
   // Import all dependencies
   import flash.display.MovieClip;
   import flash.events.MouseEvent;
   // Declare our class
   public class MediaPlayer extends MovieClip
   {
      // Constructor function.
      // Called when someone creates a new MediaPlayer
      public function MediaPlayer()
      {
         // Make sure we call the MovieClip constructor
         super();
         // Let us know that we created this player.
         trace( "MediaPlayer created!" );
      }
      // Play the media file
      public function playFile()
      {
      }
      // Pause the media file
      public function pause()
      {
      }
      // Load a media file.
      public function load( file:String )
      {
      }
   }
}
```

Now that the control bar has been removed from the `MediaPlayer` class, our next step is to re-add the `ControlBar` to the stage when the player runs.

Adding the ControlBar to the stage

In order to add our new `ControlBar` to the stage, we will need to make a few modifications to our `main.as` file. Our goal here is to allow our Flash application to run as a remote control application, a media player, or both. Because of this, we will need to design our Flash application so that it can handle situations where we would like it to behave only as a control bar. For this use case, we will need the ability to resize our Flash application so that it only shows the control bar section and not the media region. However, by default, Flash will scale the size of our application to fit any embedded width and height that we provide, which is not what we want. Instead, we would like the Flash application to mask off any region outside of the width and height region provided to our HTML object code. To do this, we will need to add the following code to our `main.as` file, to tell our stage to not scale when the player is being resized:

```
// Set up our responder with the callbacks.
var responder:Responder = new Responder( onConnect, onError);
// We do not want to scale the stage.
stage.scaleMode = StageScaleMode.NO_SCALE;
stage.align = StageAlign.TOP_LEFT;
// Connect to Drupal
drupal.call("system.connect", responder);
```

Our next task is to add our `ControlBar` to the stage. Again, we need to be able to handle the use case where our player will be used as a remote control only. For this case we do not really require the `nodeId` variable to be valid, since we can connect to any remote media player and let them worry about the `nodeId` of the media to play. For this reason, we will need to add our `ControlBar` in two different places. The first place will be within the `onNodeLoad` function, when a node has successfully been added to the player; and the second place is when a `nodeId` is not provided. These changes look like the following:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
   // Declare our base URL.
   var fileURL:String = baseURL;
   // Add our file's relative path.
   fileURL += "/";
   // Declare a generic media player.
   var player = null;
   // If this node has audio.
   if( node.audio ) {
      // Declare our player as an AudioPlayer.
      fileURL += node.audio.file.filepath;
      player = new AudioPlayer();
   }
   else if( node.field_video ) {
      // Declare our player as a VideoPlayer.
      fileURL += node.field_video[0]["filepath"];
      player = new VideoPlayer();
   }
   // Add the player to the stage.
   addChild( player );
   // Add a control bar.
   addControlBar();
   // Play our audio file
   player.load( fileURL );
}
// Add the control bar to the stage.
function addControlBar()
{
   var controlBar:ControlBar = new ControlBar();
   addChild( controlBar );
}
// Called when Drupal returns with a successful connection.
function onConnect( result:Object )
{
   // Set our sessionId variable.
   sessionId = result.sessid;
   trace("We are connected!!!");
   trace("Session Id: " + sessionId);
   // If the node Id is valid...
   if( nodeId ) {
   // Load our node.
   loadNode( nodeId );
   }
   else {
   // Add a control bar.
   addControlBar();
   }
}
```

Now that we have removed the control bar dependency from the media player, we must recreate the communication channels between `ControlBar` and `MediaPlayer` by building a new communication mechanism that will support both local and remote connections.

Communication between ControlBar and MediaPlayer

In a typical Flash architecture, communication between two different components would use ActionScript's event model to dispatch events from one component to the next. But, for our implementation, we will need to take an unconventional approach with our architecture, since the communication between these two different components can be performed, either locally or remotely, between two separate Flash applications. Because of this, we will need to construct a gateway class whose sole purpose is to link the `ControlBar` to the `MediaPlayer` either directly or remotely. Although this method is not the only way to do this type of interaction, it can be argued that it gives maximum flexibility, and allows us to have the same code to control a remote or a local connection between the two components. We can see an illustration of this by constructing the `gateway` class.

Creating a communication gateway

Our `gateway` class will simply be a new class that serves as a communication gateway between the `ControlBar` and the `MediaPlayer`. To start with, we will create a new file called `MediaGateway.as` and open up that file within our Flash IDE. One major difference between this class and the other classes we have implemented in this book is that this gateway should be declared and used statically.

Using static functions

A static function is a special type of function, within a class, which can be called without the requirement for an instantiated object. What this means is that any object can access each function declared within our `MediaGateway` class by calling the function directly, instead of calling through a declared object. For example, if we wish to use the `MediaPlayer` class to play a media track, we must first declare a `MediaPlayer` object and then load that media file by calling a non-static function on that media object as follows:

```
var media:MediaPlayer = new MediaPlayer();
media.load( "mymusic.mp3" );
```

By declaring a function as static, we are saying that the functionality, within that function, is common for all instances of any declared object. We could call that function as if it were a global function by simply calling the function within the class, without declaring an instance of that class as follows:

```
MediaPlayer.load( "mymusic.mp3" );
```

The main drawback of having statically defined functions is that only one instance of that functionality can exist. What this means is, for this example, we would not be able to declare two separate instances of `MediaPlayer` and have them behave independently of each other. For our gateway, however, this is not an issue since its objective is to only serve as a message router between the `ControlBar` and `MediaPlayer` classes. With that said, we can now create a new class that has four static functions, which will trigger the play and pause events for both the `ControlBar` and `MediaPlayer` classes. The shell for this class should look something similar to the following:

```
package
{
   // Declare our class
   public class MediaGateway
   {
      // Add a static play function
      public static function playMedia()
      {
         // TO-DO:  Play the media
      }
      // Add a static pause function
      public static function pauseMedia()
      {
         // TO-DO:  Pause the media
      }
```

```
         // Add a static control play function
         public static function playControl()
         {
            // TO-DO:  Play the control
         }
         // Add a static control pause function
         public static function pauseControl()
         {
            // TO-DO:  Pause the control
         }
      }
   }
```

Now that we have our gateway functions defined, our next task is to find a way to trigger the non-static functions of the `MediaPlayer` and `ControlBar` instances. When devising a plan to incorporate both the `MediaPlayer` and `ControlBar` instances, within our static gateway, we run into a unique obstacle in that non-static objects and variables (like the `MediaPlayer` and `ControlBar`) cannot be referenced within a static function. The good news is that there is a workaround, where we can declare two static variables within the `MediaGateway` class that references both the `ControlBar` and `MediaPlayer` instances, since there should be only one of each per Flash application. We can start this association by first declaring the static variables within our `MediaGateway` class that will be used to reference the `ControlBar` and `MediaPlayer` objects.

```
   package
   {
      // Declare our class
      public class MediaGateway
      {
         // Add a static play function
         public static function playMedia()
         {
            // TO-DO:  Play the media
         }
         // Add a static pause function
         public static function pauseMedia()
         {
            // TO-DO:  Pause the media
         }
         // Add a static control play function
         public static function playControl()
         {
            // TO-DO:  Play the control
         }
```

```
    // Add a static control pause function
    public static function pauseControl()
    {
      // TO-DO:  Pause the control
    }
    // Declare our obect references.
    public static var controlBar:ControlBar;
    public static var mediaPlayer:MediaPlayer;
  }
}
```

To complete this association, we need to set these static references to the non-static objects they represent. We can do this, within each class, by assigning the correct static object to the `this` pointer of the `MediaPlayer` and `ControlBar` instances.

Using the this pointer

The `this` pointer is a special pointer that is used, within any class, as a reference to itself. We can use the `this` pointer to make the association between the static variables within our `MediaGateway` and the actual `ControlBar` and `MediaPlayer` objects. The only problem here is that we need to find a good spot to make this association. We cannot do it within the constructor of each class since the `this` pointer has not been instantiated at the point of object construction. We can, however, trigger when the object has been added to the stage, as and when someone uses the `addChild` method to add that object to the stage, and then make that association within the handler function. We can do this by triggering on the `ADDED_TO_STAGE` event, and then assigning the static objects to the `this` pointer once that object has been added to the stage. We are guaranteed, at that point, to have a valid `this` pointer to make the correct association. Our `MediaPlayer` and `ControlBar` classes should now look like the following:

MediaPlayer.as

```
// Import all dependencies
import flash.events.Event;
import flash.display.MovieClip;
import flash.events.MouseEvent;
// Declare our class
public class MediaPlayer extends MovieClip
{
  // Constructor function.
  // Called when someone creates a new MediaPlayer
  public function MediaPlayer()
  {
     // Make sure we call the MovieClip constructor
     super();
     // Let us know that we created this player.
     trace( "MeidaPlayer created!" );
     // Trigger when this object is added.
     addEventListener( Event.ADDED_TO_STAGE, onAdded );
  }
  // Called when this object has been added to the stage.
  private function onAdded( event:Event )
  {
     // Assign this object as the active media player.
     MediaGateway.mediaPlayer = this;
  }
...
...
```

ControlBar.as

```
// Import all dependencies
import flash.events.Event;
import flash.display.MovieClip;
import flash.events.MouseEvent;
// Declare our class
public class ControlBar extends MovieClip
{
   // Constructor function.
   // Called when someone creates a new ControlBar
   public function ControlBar()
   {
      // Make sure we call the MovieClip constructor
      super();
      // Let us know that we created the ControlBar.
      trace( "ControlBar created!" );
      // Setup the play button.
      playButton.buttonMode = true;
      playButton.mouseChildren = false;
      playButton.addEventListener(MouseEvent.MOUSE_UP,onPlay);
      // Setup the pause button.
      pauseButton.buttonMode = true;
      pauseButton.mouseChildren = false;
      pauseButton.addEventListener(MouseEvent.MOUSE_UP,onPause);
      // Set the state of the play and pause buttons.
      // We want to show the play button at first, so...
      playButton.visible = true;
      pauseButton.visible = false;
      // Trigger when this object is added.
      addEventListener( Event.ADDED_TO_STAGE, onAdded );
   }
   // Called when this object has been added to the stage.
   private function onAdded( event:Event )
   {
      // Assign this object as the active control bar.
      MediaGateway.controlBar = this;
   }
...
...
```

Making the connections

After we are done with this crucial step, we can now complete our `MediaGateway` class to call the `playFile` and `pause` functions on our `mediaPlayer` and `controlBar` objects.

```
package
{
   // Declare our class
   public class MediaGateway
   {
      // Add a static play function
      public static function playMedia()
      {
         // Play the media
         mediaPlayer.playFile();
      }
      // Add a static pause function
      public static function pauseMedia()
      {
         // Pause the media
         mediaPlayer.pause();
      }
      // Add a static control play function
      public static function playControl()
      {
         // Play the control
         controlBar.playFile();
      }
      // Add a static control pause function
      public static function pauseControl()
      {
         // Pause the control
         controlBar.pause();
      }
      // Declare our obect references.
      public static var controlBar:ControlBar;
      public static var mediaPlayer:MediaPlayer;
      }
   }
```

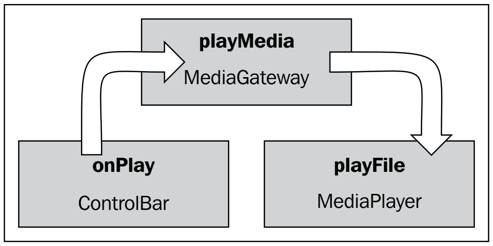
After we are done with this, our final step is to change the onPlay and onPause functions within our ControlBar.as file, to use the MediaGateway calls to play and pause the media.

```
   // Called when the play button is pressed.
   private function onPlay( event:MouseEvent )
   {
      // Tell the player to play
      MediaGateway.playMedia();
   }
   // Called when the pause button is pressed
   private function onPause( event:MouseEvent )
   {
      // Tell the player to pause
      MediaGateway.pauseMedia();
   }
```

We can now also change the playFile and pause functions, within the MediaPlayer.as file, to make a call to the MediaGateway's playControl and pauseControl to set the correct state of our ControlBar.

```
   // Play a media file.
   public function playFile()
   {
      // Tells the media gateway to play the controls.
      MediaGateway.playControl();
   }
   // Pause a media file
   public function pause()
   {
      // Tells the media gateway to pause the controls.
      MediaGateway.pauseControl();
   }
```

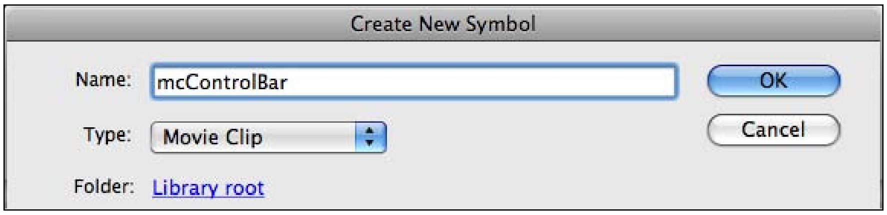
To help illustrate this unique messaging system, the following diagram shows how the ControlBar is now able to call the playFile routine of the MediaPlayer object through the use of the static MediaGateway playMedia function.



I know that this may seem like a lot of work to simply call a function on another object, but what we are doing here is setting up the foundation to create a remote connection between the `ControlBar` and the `MediaPlayer` by using a common `gateway` class that governs the communication between the two. In the next chapter we will expand this and introduce a remote communication, but for now we need to make sure the local communication works as expected. To do this, we will need to change our media player project so that it reflects the new object structure to include the `ControlBar`.

Adding the ControlBar to our Flash project

For this step, we will now open up the `chapter7.fla` file in our Flash IDE, where we will extract the `ControlBar` objects from the `MediaPlayer` movie clips. There are several ways to accomplish this step, but I will try to take the path of least resistance by first creating a new MovieClip that will be called **mcControlBar**. We can do this by clicking on the menu item **Insert | Create Symbol**.



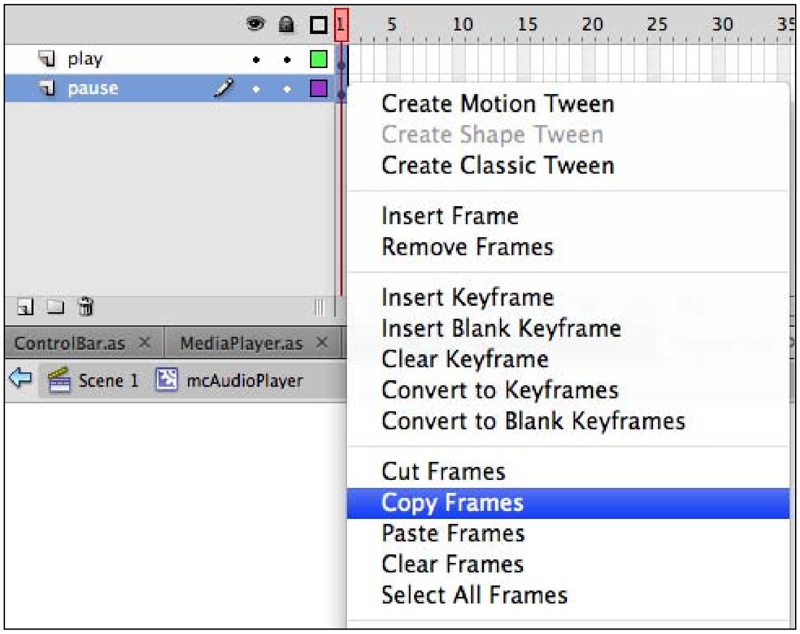
Create New Symbol

Name: mcControlBar

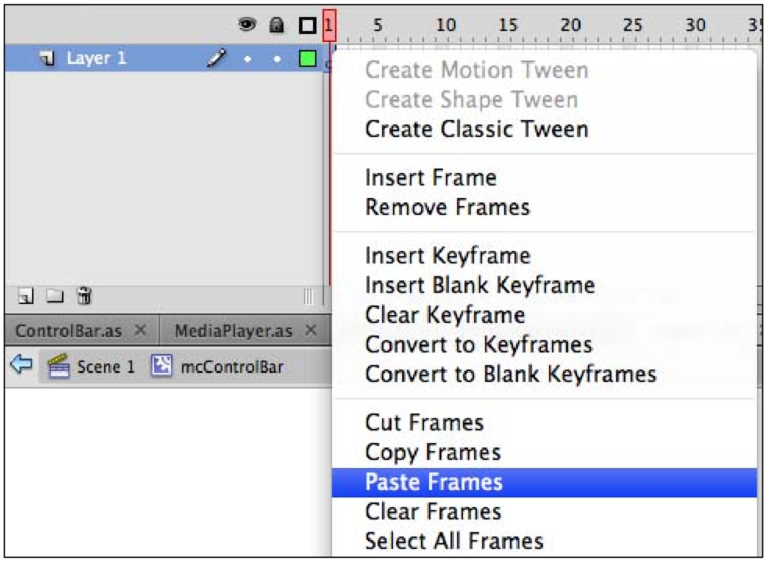
Type: Movie Clip

Folder: Library root

Once you click on **OK**, our next step will be to replicate the play and pause button functionality from either the `mcAudioPlayer` or the `mcVideoPlayer` and put that inside the `mcControlBar` movie clip. We can do this by first opening up the `mcAudioPlayer` movie clip within our Library. Once this movie clip is open, we need to make sure we unlock both the **play** and **pause** layers and then select the first key frames for the play and pause layers (by selecting one and then pressing the *Shift* key and selecting the other), and then right-click on the selection within the first key frame, and select **Copy Frames**.



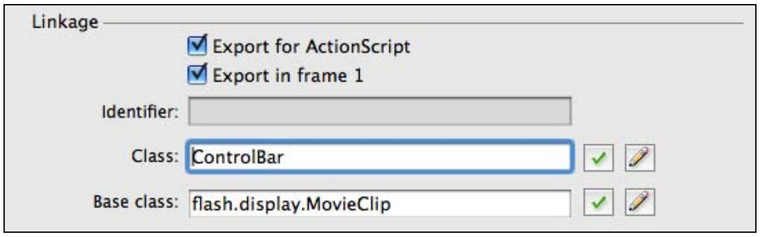
With these frames copied, we will double-click on our `mcControlBar` movie clip from our Library, select the first key frame on the default layer, right-click, and then select **Paste Frames** (as shown in the following screenshot).



After doing this, we will see that the layers, frames, and all the content have been copied to the `mcControlBar` movie clip.

Now that we have a complete control bar movie clip, we need to make sure that this movie clip will reference the `ControlBar` class that we just created in ActionScript. We can do this by shifting our focus to our Flash Library, right-clicking on the `mcControlBar` movie clip, and then selecting **Properties**.

Much like what we did with the `mcAudioPlayer` and the `mcVideoPlayer`, we need to make sure we check **the Export for ActionScript** checkbox, and then provide **ControlBar** for the class as shown:



Linkage

☑ Export for ActionScript

☑ Export in frame 1

Identifier:

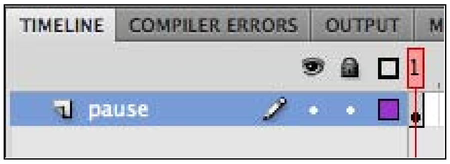
Class: ControlBar

Base class: flash.display.MovieClip

After clicking on **OK**, we are ready to modify our `mcAudioPlayer` and `mcVideoPlayer` movie clips so that they do not contain any control bar functionality.

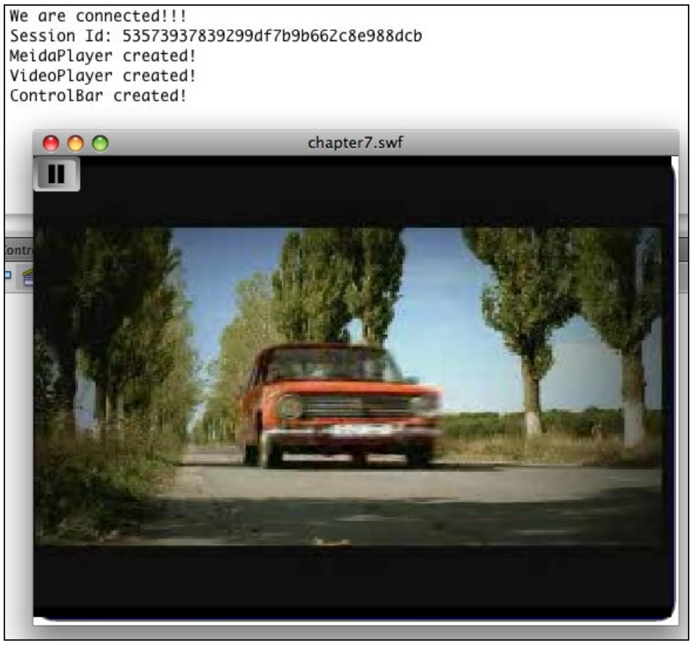
Removing the control bar from the MediaPlayer

Our next task is to open up both the `mcAudioPlayer` and `mcVideoPlayer` movie clips and remove the control bar functionality from them. This can easily be done by removing the play and pause layers within each one of these movie clips. We will start by opening up the `mcAudioPlayer` movie clip, where we will then delete the play layer leaving only the pause layer.



Since the **pause** layer is the only layer left, Flash will not let us delete that layer. We can, however, select the contents of the first key frame of this layer and delete the contents. We can do this by clicking on the **pause** layer and then pressing the *Delete* key on our keyboard. Now that we have cleaned out the Audio Player, we can do the same to the `mcVideoPlayer`, so that there are no more remnants of the control bar in either of these movie clips.

At this point, we should be able to run our application and see that it functions as it did at the end of the previous chapter, but this time using an abstracted `ControlBar` with the foundations of a `MediaGateway` in place.



```
We are connected!!!
Session Id: 53573937839299df7b9b662c8e988dcb
MeidaPlayer created!
VideoPlayer created!
ControlBar created!
```

In the next chapter, our task will be to separate the `ControlBar` and `MediaPlayer` functionalities between two separate Flash applications, essentially creating a Remote Control application, and then integrate those two applications into our Drupal web site.

Summary

In this chapter we learned the basics of how to take an existing Flash application and break apart the components for remote communication. We achieved this by first abstracting out separate functionalites into two separate components, and then laying the foundation for a communication gateway between the two different components. This is an essential first step to create a robust and easily maintained system, where Flash applications can be separated on a Drupal web site, thus implementing a hybrid Flash integration approach.

In the next chapter, we will pick up right where we left off with this chapter by taking our abstracted `ControlBar` and `MediaPlayer` and implementing the remote communication needed to separate the two Flash applications on our Drupal web site.

8

The Hybrid Approach

Part 2: Remote Control

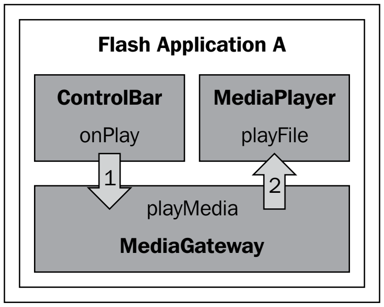
We have already extracted all of our control functionality from our media player, and in this chapter we will focus on creating the bridge between two different Flash applications. Once we create this bridge, we will have the ability to control our media from a remote Flash application. In other words, we will be building a remote control for our media player that can be placed anywhere on the page, separate from the media player. We will achieve this separation by walking through the following steps:

- Client-side Flash communication
- Flash to JavaScript communication
- Creating a JavaScript gateway
- Flash and JavaScript synchronization
- Using our remote control within Drupal

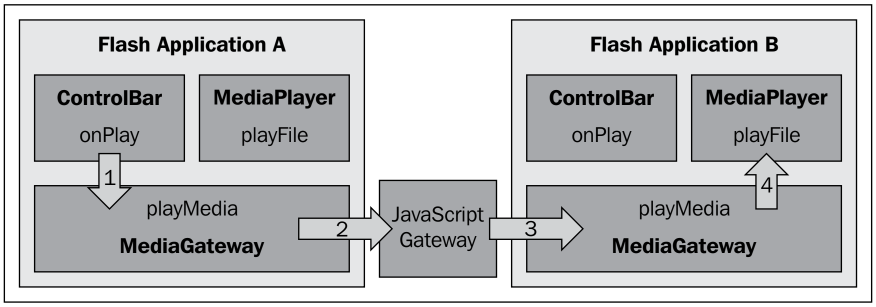
Client-side Flash communication

In the previous chapters we learned how Flash communicates with Drupal using a client-server communication called web services. This type of communication allowed for our Flash applications to extract Drupal content to be used in our Flash applications by calling routines located on the server. In this chapter, we will shift our focus away from the server-side and concentrate on client-to-client communication. We will build the communication gateway between Flash and JavaScript that will allow us to interconnect separate Flash applications on a single page. By connecting Flash applications together using JavaScript, we will have a componentized architecture, where each Flash "widget" on our web site has a specific function, while at the same time, works with other widgets to achieve a unified user interface. This will give us much flexibility while developing our Flash user interfaces for Drupal. Let me illustrate how this works.

Picking up where we left off from the previous chapter, we abstracted out a `ControlBar` class that receives the button pressed from the user, and then calls the `MediaGateway` `playMedia` function, which would then just call the `MediaPlayer` `playFile` function. All of the communication between the `ControlBar` to the `MediaPlayer` classes are performed locally, since this path of communication is handled within a single Flash application as follows:



Let's now illustrate the architecture if we were to break apart this functionality into two different Flash applications. One application is responsible for controlling the media (the remote control), while the other application is responsible for playing the media (the player). This is where the `MediaGateway` becomes useful. Since we created a separate component to handle the communication between the `ControlBar` and the `MediaPlayer`, we can now add to the functionality of this gateway to allow remote communication. What is even more important to note is that this `MediaGateway` can also be configured to call the function directly within the same Flash application as we did in the previous chapter. By designing our Flash application such that it can have either remote or local communication, we are giving it maximum flexibility so that the functional components within our application can be either combined or separated on our web site.



Our goal for this chapter will be to take our media player project from the previous chapter and create this type of architecture. But first, we will need to modify our `MediaGateway.as` file so that it can communicate to an external JavaScript gateway.

Flash to JavaScript communication

Before we begin this section, let's copy the project and source files from the previous chapter into a new folder. We will use this folder to build the project for this chapter. After we have done that, we can now modify our `MediaGateway.as` file so that it can communicate with an external JavaScript gateway. In order to allow Flash to communicate with JavaScript, we will use a standard Flash component called `ExternalInterface`.

Calling a JavaScript function from Flash

The `ExternalInterface` component allows us to register functions that can be called from JavaScript as well as gives us the ability to call any JavaScript function directly from within Flash. This Flash to JavaScript communication can be achieved by using the `ExternalInterface.call` function. This function takes several different arguments, where the first argument is the name of the function you would like to call in JavaScript, and any arguments following that are then passed along to that JavaScript function as its arguments. For example, let us suppose that we wish to call a JavaScript function called `printAlert`, which has only a single argument that will be passed to an `alert` function. In JavaScript, this function will look like the following:

```
function printAlert( alertText )
{
   alert( alertText );
}
```

We can use `ExternalInterface` from our Flash application to call this external JavaScript function by making the following call within Flash:

```
ExternalInterface.call("printAlert", "Hello World!");
```

If we were to build this Flash application and place it on a web site next to the JavaScript shown above, we would see a message window with the text `Hello World!` shown within that window.

Not only can the `ExternalInterface` component be used to communicate from Flash to JavaScript, it also allows communication in the other direction, where JavaScript can communicate to Flash.

Calling a Flash function from JavaScript

We will now explore the reverse functionality, from the previous section, by using the `ExternalInterface` component to accept incoming function calls from JavaScript. The process for allowing incoming messages requires an extra step in which Flash should first register each externally public function as "callable". This can be done using the `ExternalInterface.addCallback` function, where the first argument is the name of the function that JavaScript will call to instantiate the function, and the second argument is the actual Flash function that gets called with that callback. As an example of this, let's suppose that we want JavaScript to trigger a trace statement in Flash with some specified text. On the Flash side, we will first need to register a `callTrace` function by using `ExternalInterface.addCallback` followed by the declaration of the function being called. Also, note that these are only examples and should not be placed in the `MediaGateway` file.

```
ExternalInterface.addCallback("callTrace", callTrace);

public static function callTrace( traceText:String )
{
   trace( traceText );
}
```

By adding the callback to our `ExternalInterface`, we are now allowing JavaScript to call the `callTrace` function on the Flash object embedded in our HTML page. Although the JavaScript side of this communication is still unclear, we will need to put that off for a later section in this chapter. For now, we will concentrate on the Flash side so that we can modify our `MediaGateway` for external communication, and then complete the puzzle when we build our JavaScript gateway later in this chapter.

So, let's move on to the implementation of the `ExternalInterface` into our `MediaGateway` component. Our first task, in this endeavor, will be to initialize `ExternalInterface` before it is used in our Flash application.

Initializing the ExternalInterface

For this section we will start out by opening up the `MediaGateway.as` file that we created in the previous chapter. At the point where we left off, this file should look like the following:

```
package
{
   // Declare our class
   public class MediaGateway
   {
      // Add a static play function
      public static function playMedia()
```

```
        {
           // Play the media
           mediaPlayer.playFile();
        }
        // Add a static pause function
        public static function pauseMedia()
        {
           // Pause the media
           mediaPlayer.pause();
        }
        // Add a static control play function
        public static function playControl()
        {
           // Play the control
           controlBar.playFile();
        }
        // Add a static control pause function
        public static function pauseControl()
        {
           // Pause the control
           controlBar.pause();
        }
        // Declare our obect references.
        public static var controlBar:ControlBar;
        public static var mediaPlayer:MediaPlayer;
     }
  }
```

Our first goal will be to create an initialize function that will be used to register any external functions that will be used by our JavaScript gateway. For this chapter, the only required external functions will be the four static functions that we defined in our `MediaGateway`. After this has been done, we can then trigger a callback telling whoever called this initialize function that they can continue with their initializing process.

We will also need to make sure that we handle the ID of the Flash application that we would like to connect to for external communication. What this implies is that each time we declare an HTML `<object>` for a Flash application, we will need to give it an ID so that the JavaScript Gateway can locate the Flash application that will be receiving messages.

The changes to add to our initialize routine should look as follows:

```
package
{
   // Import all necessary components.
   import flash.external.ExternalInterface;
   // Declare our class
   public class MediaGateway
      // Used to initialize the MediaGateway.
      public static function initialize( _id:String, _connect:String,
      onLoaded:Function )
      {
         // Setup our gateway variables.
         id = _id;
         connect = _connect;
         onLoaded = _onLoaded;
         // See if the ExternalInterface is available.
         if ( connect && ExternalInterface.available ) {
            ready = true;
            // Register for all our JavaScript callbacks.
            ExternalInterface.addCallback("playMedia", playMedia);
            ExternalInterface.addCallback("pauseMedia", pauseMedia);
            ExternalInterface.addCallback("playControl",playControl);
            ExternalInterface.addCallback("pauseControl",
            pauseControl);
            // Call our callback function.
            onLoaded();
         }
         else {
            ready = false;
            onLoaded();
         }
      }
      // Add a static play function
      public static function playMedia()
      {
         // Play the media
         mediaPlayer.playFile();
      }
      // Add a static pause function
      public static function pauseMedia()
      {
         // Pause the media
         mediaPlayer.pause();
      }
      // Add a static control play function
      public static function playControl()
      {
         // Play the control
```

```
                controlBar.playFile();
            }
            // Add a static control pause function
            public static function pauseControl()
            {
            // Pause the control
                controlBar.pause();
            }
            // Declare our obect references.
            public static var controlBar:ControlBar;
            public static var mediaPlayer:MediaPlayer;
            // Declare our private variables.
            private static var id:String;
            private static var connect:String;
            private static var onLoaded:Function;
            private static var ready:Boolean;
        }
    }
```

Now that we have this initialize routine, we can plug this into our normal initialization process within our core `main.as` file.

Adding the MediaGateway initialization to main.as

Our next task will be to open up the `main.as` file, where we will include the initialization of the `MediaGateway` in our boot up sequence. Here, our goal will be to find the right spot to include this initialization, while at the same time, keep a synchronous boot up process. By synchronous, I mean that we should not start a new boot up process unless the previous process has finished. For the `MediaGateway` initialize routine, this will require us to take advantage of the `callback` function to let us know when our gateway has finished initializing.

Through observation of the `main.as` file, we can determine that a logical place for the `MediaGateway` initialization should probably occur before the connection with Drupal has started. Because of this, we should make a change to call `system.connect` after the media gateway has finished initializing. This change can be realized by simply placing the `system.connect` call within its own function, and then using that function as the callback pointer to our `MediaGateway.initialize` routine. Then, we can use FlashVars to pass in the correct value for the `id` and the `connect` parameters needed to make our Gateway connection.

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";
var nodeId:Number = root.loaderInfo.parameters.node;
var id:String = root.loaderInfo.parameters.id;
var connect:String = root.loaderInfo.parameters.connect;
...
...
// Connect to Drupal
function connectToDrupal()
{
   drupal.call("system.connect", responder);
}
// Initialize the MediaGateway
MediaGateway.initialize( id, connect, connectToDrupal );
```

Although this change may seem trivial, we can see that the boot order has completely been changed by simply moving the Drupal `system.connect` call within a function, and then adding that function as the callback function for our `MediaGateway` initialize routine. Now, the `system.connect` will only occur when the `MediaGateway` has finished initializing.

Now that we have all the information we need to connect to another application, we can shift our focus back to the `MediaGatway`, where we will create the outgoing messages to JavaScript.

Adding outgoing messages to the MediaGateway

One important thing to note about the `MediaGateway` is that it is responsible for all incoming and outgoing messages for our Flash application. We covered any incoming messages by registering for the callback functions using the `ExternalInterface`, but one thing we have not covered yet is the outgoing messages. If we focus solely on the `MediaGateway`, we can now see how it behaves as a message router for our Flash application. It should be responsible for internal and external interaction by handling messages from JavaScript as well as internal function calls. Fortunately, the amount of code needed to realize this is minimal, since we can expand our current static gateway functions to also include the external interaction needed to complete this communication cycle.

Since we will be calling these functions internally, we need a way to call them differently—whether they are called from a remote location or internally. We can do this by utilizing default arguments for all of our static gateway functions as follows:

```
// Add a static play function
public static function playMedia(external:Boolean = false)
{
   // Play the media
   mediaPlayer.playFile();
}
// Add a static pause function
public static function pauseMedia(external:Boolean = false)
{
   // Pause the media
   mediaPlayer.pause();
}
// Add a static control play function
public static function playControl(external:Boolean = false)
{
   // Play the control
   controlBar.playFile();
}
      // Add a static control pause function
public static function pauseControl(external:Boolean = false)
{
   // Pause the control
   controlBar.pause();
}
```

This will work because when our JavaScript gateway makes any call to one of our gateway functions, it will not provide any argument with that call. Since any call from an external location will always need to be routed to an internal location, we are forcing the local function to be called when the message comes from an external source. Although the outgoing messages have not been implemented, we can see how we can control which messages are external and which ones will be internal by simply providing the correct Boolean value to these gateway functions.

We can now modify our static gateway functions so that they will call the `ExternalInterface` if the variable **external** is set to true, and make an internal call if the variable **external** is set to false. We also need to verify that both the **connect** and **ready** flags are set to true before making any `ExternalInterface` calls. This change will make our four gateway functions look like the following:

```
// Add a static play function
public static function playMedia( external:Boolean = false )
{
   // Check to see if an external call should be made.
   if( external && connect && ready ) {
      // Call the JavaScript playMedia function.
      ExternalInterface.call( "playMedia", connect );
   }
   else {
      // Play the media
```

```
         mediaPlayer.playFile();
      }
   }
         // Add a static pause function
   public static function pauseMedia( external:Boolean = false )
   {
      // Check to see if an external call should be made.
      if( external && connect && ready ) {
      // Call the JavaScript pauseMedia function.
      ExternalInterface.call( "pauseMedia", connect );
      }
      else {
         // Pause the media
         mediaPlayer.pause();
      }
   }

   // Add a static control play function
   public static function playControl( external:Boolean = false )
   {
      // Check to see if an external call should be made.
      if( external && connect && ready ) {
         // Call the JavaScript playControl function.
         ExternalInterface.call( "playControl", connect );
      }
      else {
         // Play the control
         controlBar.playFile();
      }
   }
   // Add a static control pause function
   public static function pauseControl( external:Boolean = false )
   {
      // Check to see if an external call should be made.
      if( external && connect && ready ) {
         // Call the JavaScript pauseControl function.
         ExternalInterface.call( "pauseControl", connect );
      }
      else {
         // Pause the control
         controlBar.pause();
      }
   }
```

The last step in this process will be to modify our `ControlBar.as` and `MediaPlayer.as` files so that they will always attempt remote communication.

Adding remote or local functionality

Since the ControlBar and MediaPlayer components are responsible for calling the MediaGateway routines, we will need to modify them so that they will provide the Boolean flags necessary to tell the MediaGateway that this call was made locally. We can easily achieve our goals by always passing true for the external parameter, and then letting our MediaGateway decide if a remote call should really be made. Within the ControlBar.as and MediaPlayer.as files, these changes will look as follows:

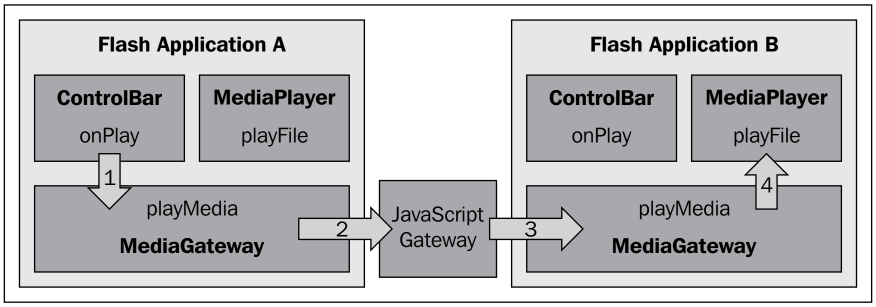
ControlBar.as

```
// Called when the play button has been pressed.
private function onPlay( event:MouseEvent )
{
   // Tell the player to play.
   MediaGateway.playMedia( true );
}
// Called when the user presses the pause button.
private function onPause( event:MouseEvent )
{
   // Tell the player to pause.
   MediaGateway.pauseMedia( true );
}
```

MediaPlayer.as

```
// Play a media file
public function playFile()
{
   // Tells the media gateway to play the controls.
   MediaGateway.playControl( true );
}
// Pause a media file
public function pause()
{
   // Tells the media gateway to pause the controls.
   MediaGateway.pauseControl( true );
}
```

At this point, our code can be used by both internal and external components and will route them accordingly.



From looking at this diagram, we can see that the missing piece in this puzzle is the JavaScript Gateway to transfer the messages to and from each Flash application.

Building a JavaScript Gateway

For this section, we will now concentrate our attention on JavaScript, where we will create the gateway needed to complete the message transfer between each Flash application. We will begin by creating a new empty file, within our `chapter8` directory, called `gateway.js` and then opening that file in your favorite text editor. Our goal here will be to receive the messages from any Flash application, and then locate the other Flash application using the *connect* ID, and then call the same function within that application. We will start this process by locating a Flash application using JavaScript.

Locating a Flash application using JavaScript

For this section we will construct a function that will be used to locate any Flash application given the ID of the object used to embed that application. This can be done using the `document` element within JavaScript, and then indexing the Flash application using the ID passed to our new function. Unfortunately, we will also need to add compatibility for other browsers, ahem… Internet Explorer! This function should look like the following within our `gateway.js` file:

```
/**
 *  Returns a Flash Object given an ID
 *
 *  @param - The ID of the player.
 */
function getFlashObject( id )
{
      var flashObj = null;
      // Check for Internet Explorer
      if(navigator.appName.indexOf("Microsoft") != -1) {
         flashObj = window[id];
      }
      else {
         // Index our ID within the document object
         if(document[id].length != undefined) {
            flashObj = document[id][1];
         }
         else {
            flashObj = document[id];
         }
      }
      // We want to alert them if the object was not found.
      if( !flashObj ) {
         alert( id + " not found!" );
      }
      return flashObj;
   }
```

Our next task will be to create the gateway functions that will utilize this function to tie two Flash applications together.

Creating the gateway functions between two Flash applications

Now that we have a function that can locate any Flash object embedded on our page, we can construct our gateway functions. These gateway functions will be responsible for accepting a function call from one Flash application, and then locating and calling the same function on another Flash application. These gateway functions should be labeled the same as they are within our `MediaGateway`, and are implemented as follows in our `gateway.js` file.

```
   /**
    *  Plays the media a player.
    *
    *  @param - The id of the Flash object.
    */
   function playMedia( id )
   {
      // Find the Flash Object.
      var flashObj = getFlashObject( id );

      // Play the flash object.
      if( flashObj ) {
         flashObj.playMedia();
      }
   }

   /**
    *  Pauses the media a player.
    *
    *  @param - The id of the Flash object.
    */

   function pauseMedia( id )
   {
      // Find the Flash Object.
      var flashObj = getFlashObject( id );

      // Pause the flash object.
      if( flashObj ) {
         flashObj.pauseMedia();
      }
   }

   /**
    *  Plays the control of a player.
    *
    *  @param - The id of the Flash object.
    */

   function playControl( id )
   {
      // Find the Flash Object.
      var flashObj = getFlashObject( id );

      // Play the flash object.
      if( flashObj ) {
         flashObj.playControl();
      }
   }
```

```
/**
 *  Pauses the control of a player.
 *
 *  @param - The id of the Flash object.
 */
function pauseControl( id )
{
   // Find the Flash Object.
   var flashObj = getFlashObject( id );
   // Pause the flash object.
   if( flashObj ) {
      flashObj.pauseControl();
   }
}
```

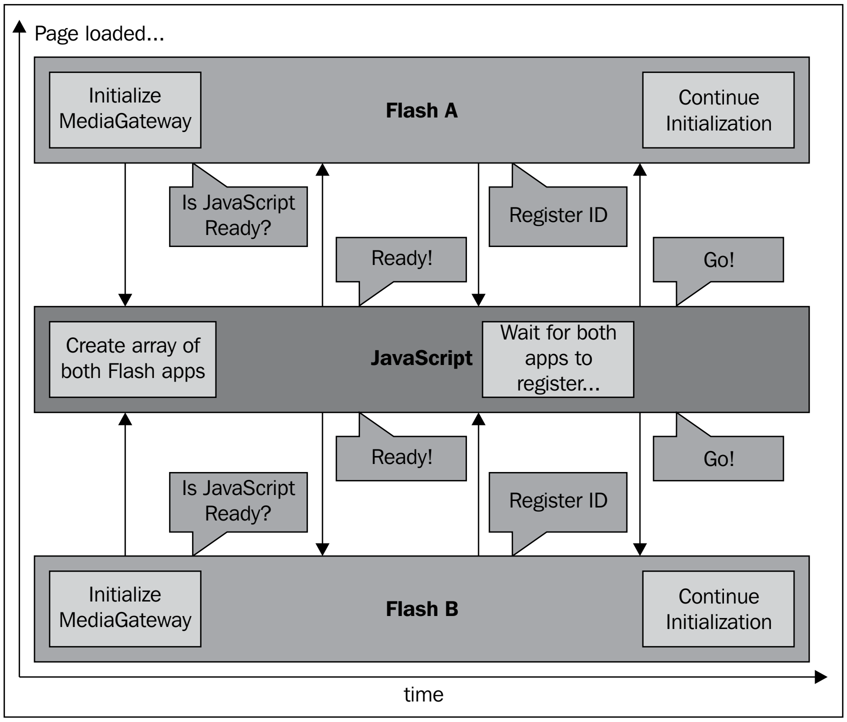
At this point, it could be argued that we have a completely functional JavaScript Gateway, since we now have a complete messaging system between two Flash applications. However, there is still a use case that will come back to bite us when implementing this gateway on a live site.

When our web page loads, there is no guarantee that one Flash application will have completed its load process before this JavaScript gateway is used from the other Flash application. This is another situation where a race condition can occur, where one Flash application loads and attempts to use the gateway before the other Flash application is ready to accept messages. It is also possible that JavaScript will not even be ready by the time that each Flash application tries to register their `ExternalInterface` functions. Because of this, we will need to modify our design to synchronize each component before they are used.

Flash and JavaScript synchronization

In order to synchronize Flash with JavaScript, we will need to modify both our `MediaGateway.as` and `gateway.js` files, so that they work together to synchronize all the components in the communication cycle. To create this synchronization, we will use some handshaking methods, where the `MediaGateway` will communicate to our JavaScript Gateway to make sure that the other Flash application is ready to go before continuing with the initialization process.

To help illustrate this communication, it is beneficial to show a time graph of how both Flash applications and JavaScript communicate shortly after a page has loaded. Hopefully, this will illustrate the chain of events that need to occur in order for our Flash application to guarantee that the other Flash application is "present and accounted for".



This diagram illustrates the necessary steps needed to make sure that each Flash application has initialized before they continue with their load process. We can also break this process up into a series of steps that we will implement separately to clarify this load process a little better. These steps are as follows:

Step 1: Create an array of communicating Flash applications

For this step, we will need to open up our `gateway.js` file, where we will create an array that will be used to store all the communicating Flash applications. We can accomplish this by adding the following code to the top of our `gateway.js` file.

```
// Variable to store our Flash objects.
var flashObjects = new Object;
```

We will also need to make sure that we create an easy-to-use function that we will use to add each Flash object. We can use the ID of the player as the index for the `flashObjects` array to allow for easy access when it is registered. Because of this, we will also need to provide a `ready` variable that will be used to indicate if the player has registered. This function can be defined as follows:

```
/**
 * Adds a Flash object to the list
 *
 * @param - The id of the Flash object.
 */
function addFlashObject( id ) {
   flashObjects[id] = new Object;
   flashObjects[id].ready = false;
}
```

This function will now be used, within the HTML of each page, to register the Flash applications that will communicate. For example, we can define a remote control and player by adding them within the HTML code as follows (we will actually do this in Drupal later in this chapter).

```
<script type="text/javascript">
   addFlashObject ( 'remote' );
   addFlashObject ( 'player' );
</script>
```

Step 2: Flash calls to see if the JavaScript Gateway is ready

Our next task is to make sure that Flash calls the JavaScript Gateway to make sure it is ready to go. We can start this by creating some functionality within our gateway to allow this interaction. To start, we will simply add a new variable to the top of our `gateway.js` file called `gatewayReady`, and then set it to true when the document has finished loading. Since we will be implementing this in Drupal, we can use the jQuery method to perform this check.

```
// Variable to store our Flash objects.
var flashObjects = new Object;
var gatewayReady = false;
/**
 * Checks to make sure the document is ready
 * and then sets our gateway as ready.
 */
$(document).ready(function() {
   gatewayReady = true;
});
```

Our next step, after this, will be to create a gateway function that Flash will use to see if the gateway is ready, which will simply return if the `gatewayReady` variable is true or not.

```
/**
 * Checks to make sure the document is ready
 * and then sets our gateway as ready.
 */
$(document).ready(function() {
   gatewayReady = true;
});
/**
 * Checks to see if the gateway is ready.
 */
function isGatewayReady() {
   return gatewayReady;
}
```

At this point, we can shift our focus to Flash, where we will modify our `MediaGateway.as` file to check and see if the JavaScript Gateway is ready. Since we cannot guarantee that this function will pass on the first call, we will need to set up some type of mechanism to re-check a given interval for each time it fails. We can use the `Timer` component to add this interval functionality, and even add a fail safe by adding a retry counter to continue initialization if the number of retries reaches a certain value.

All of our changes should look like the following;

```
// Import all necessary components.
import flash.external.ExternalInterface;
import flash.utils.Timer;
import flash.events.TimerEvent;
...
...
// Register for all our JavaScript callbacks.
ExternalInterface.addCallback("playMedia", playMedia);
ExternalInterface.addCallback("pauseMedia", pauseMedia);
ExternalInterface.addCallback("playControl", playControl);
ExternalInterface.addCallback("pauseControl", pauseControl);
// Is our gateway ready.
if( ExternalInterface.call("isGatewayReady") ) {
   // Continue the load process.
   onLoaded();
}
else {
   retries = 0;
   // Recheck every 100 ms.
   readyTimer = new Timer( 100 );
   readyTimer.addEventListener( TimerEvent.TIMER, checkGateway );
   readyTimer.start();
}
...
...
// Function to check for the gateway at a given interval.
private static function checkGateway(event:TimerEvent):void
{
   // Is our gateway ready.
   if( ExternalInterface.call("isGatewayReady") && (retries++ < 5) ) {
      onLoaded();
   }
}
...
...
// Declare our private variables.
private static var connect:String;
private static var onLoaded:Function;
private static var ready:Boolean;
private static var retries:uint;
private static var readyTimer:Timer;
```

We are now done with this step and ready to move on.

Step 3: Flash application registers with JavaScript

Walking down our synchronization diagram, we can see that the next step in this process for Flash is to register its ID with JavaScript. Since the other application will be doing the same, we can now see how the Gateway will know when both applications are ready for communication and then send notification to both applications when this has occurred. Since we are already in our `MediaGateway`, we will make all the necessary changes on the Flash side, and then fill in the missing pieces in JavaScript shortly afterwards.

For starters, we will need to replace any call to the onLoaded callback function with another function that will be used to register the ID of the Flash application. Then, we will simply need to wait until we receive notification from JavaScript that we are ready to go, which will then trigger the onLoaded callback function. We can do this by creating the following new functions within our MediaGateway.as file.

```
// Function to register the "connect" Flash application.
private static function register() : void
{
   // Register with the JavaScript Gateway.
   ExternalInterface.call( "registerFlashObject", id );
}
// Function called from JavaScript when all systems are go!
public static function allSystemsGo()
{
   // Call our callback function to continue loading.
   onLoaded();
}
```

And we can now replace any of our previous onLoaded calls with the new register function.

```
// Is our gateway ready.
if( ExternalInterface.call("isGatewayReady") ) {
   // Register with the gateway.
   register();
}
else {
   // Recheck every 100 ms.
   retries = 0;
   readyTimer = new Timer( 100 );
   readyTimer.addEventListener( TimerEvent.TIMER, checkGateway );
   readyTimer.start();}

// Function to check for the gateway at a given interval.
private static function checkGateway(event:TimerEvent):void
{
   // Is our gateway ready.
   if( ExternalInterface.call("isGatewayReady") && (retries++ < 5) ) {
      // Register with the gateway.
      register();
   }
}
```

And finally, we cannot forget to register the allSystemsGo function, so that it can be called from JavaScript when all Flash applications have registered.

```
// Register for all our JavaScript callbacks.
ExternalInterface.addCallback("playMedia", playMedia);
ExternalInterface.addCallback("pauseMedia", pauseMedia);
ExternalInterface.addCallback("playControl", playControl);
ExternalInterface.addCallback("pauseControl", pauseControl);
ExternalInterface.addCallback("allSystemsGo", allSystemsGo);
```

Now that we have this in place, we can shift our focus to our JavaScript gateway, where we will fill in the missing pieces to implement this functionality.

Step 4: JavaScript initializes our Flash when all have registered

We can now open up our gateway.js file, and first, implement the function used to register each Flash application as they "check in". This can be done by checking if the object exists in our array, and then setting the ready flag to true if it does. This can be written as follows:

```
function addFlashObject( id ) {
   flashObjects[id] = new Object;
   flashObjects[id].ready = false;
}
/**
 * Registers a flash player object.
 */
function registerFlashObject( id ) {
   // Check if this object exists...
   if( !flashObjects[id] ) {
      addFlashObject( id );
   }
   // Set this object as ready.
   flashObjects[id].ready = true;
}
```

Our next task is to check if all Flash objects within our array have registered. We can place this functionality within its own function as follows:

```
/**
 * Checks to see if all the players have registered.
 */
function allPlayersRegistered() {
   // Initialize our registered variable.
   var registered = true;

   // Iterate through all flash objects.
   for( id in flashObjects ) {
     // AND the ready flags together.
     registered = (registered && flashObjects[id].ready);
   }

   // Return if they all have registered.
   return registered;
}
```

We can now perform this check after each time that a Flash application registers to see if all of them are "present and accounted for". If they have been, then we can iterate through all the objects and then call the `allSystemsGo` function on each Flash application. This functionality looks like the following:

```
/**
 * Checks all Flash objects for registration
 * and calls their allSystemsGo functions on
 * them if they are all ready.
 */

function checkAllObjects()
{
   // Check to see if they all registered.
   if( allPlayersRegistered() ) {
     // If so, then iterate through them and...
     for ( id in flashObjects ) {
        // Find the Flash Object.
        var flashObj = getFlashObject( id );

        // Call the allSystemsGo function.
        if( flashObj ) {
            flashObj.allSystemsGo();
        }
      }
   }
}
```

And then, our last step is to add this check every time that a player registers.

```
/**
 * Registers a flash player object.
 */

function registerFlashObject( id ) {
   // Check if this object exists...
   if( !flashObjects[id] ) {
      addFlashObject( id );
   }

   // Set this object as ready.
   flashObjects[id].ready = true;
```

```
        // Check to see if this was the last...
        checkAllObjects();
    }
```

We can now pat ourselves on the back because we should have a fully functioning gateway between two Flash applications! At this point, we will need to compile our media player application so that it is ready for Drupal. To do this, we will just need to make sure that our variables have been initialized to use FlashVars when it is embedded within Drupal.

```
// Declare our variables
var baseURL:String = "http://localhost/drupal6";
var gateway:String = baseURL + "/services/amfphp";
var sessionId:String = "";
var nodeId:Number = root.loaderInfo.parameters.node;
var id:String = root.loaderInfo.parameters.id;
var connect:String = root.loaderInfo.parameters.connect;
```

After we compile our media player, we are now ready to move on to our last task, where we will implement a remote control within our Drupal web site.

Using our remote control within Drupal

The last and final task to show off the hybrid approach is to implement our Flash player along with our JavaScript Gateway in Drupal and demonstrate how all the pieces come together. We will start by adding the JavaScript gateway to Drupal.

Adding the JavaScript Gateway to Drupal

Finally, shifting our attention back to Drupal, we will start this process by adding the JavaScript gateway to Drupal. Our first step, here, is to utilize our `gateway.js` file within our site by first copying the `gateway.js` file and then placing it in our site's template folder. For example, if you are using the Garland theme, then you should copy the `gateway.js` file to the `themes/garland` folder. Once the file is in place, we can now change our template so that it includes our new `gateway.js` file.

Although there are several ways to do this next step, I would highly recommend using a new approach introduced with Drupal 6 that allows any template to include JavaScript files by adding them to the `*.info` file included with that template. For our example, since we are using the Garland template that comes with Drupal 6, we can open up the `garland.info` file within our site's `/themes/garland` folder, and then add the following line of code to include our `gateway.js` file.

```
stylesheets[all][] = style.css
stylesheets[print][] = print.css
scripts[] = gateway.js
```

Now that we have successfully integrated our `gateway.js` file, our next step is to embed the player that we just built, so that it can demonstrate the remote functionality.

Adding our Media Player to Drupal

Moving our way back to Drupal, we will start this section by overwriting our previous media player with our new one using the FlashNode module. We can do this by going to **Administer | Content**, and then editing the Media Player Flash node that is used to hold our media player application. After we have replaced our Media Player Flash node with the new Flash application, our next step is to modify our Video template so that we can include the `id` and `connect` parameters needed for remote communication.

Changing our Content Template

In this step, we will start out by navigating to **Administer | Content Templates** and will then click on the **edit template** link next to the Video node type. Once we are within the content template for our video node type, we will start by adding the `id` and the `connect` variables to the list of FlashVars passed to the player.

```
<?php
// Get the Flash application.
$sql = "SELECT * FROM {node} WHERE nid=%d";
$flash = db_fetch_object(db_query($sql, 11));
// Load the flash node.
flashnode_load( $flash );
// Set the flashvars to the right node Id.
$flash->flashnode["flashvars"] = 'node=' . $node->nid;
$flash->flashnode["flashvars"] .= '&id=player';
$flash->flashnode["flashvars"] .= '&connect=remote';
// Add the Id of our player to the flashnode object.
$flash->flashnode["id"] = 'player';
// Show the Flash application.
print theme('flashnode', $flash->flashnode, FALSE);
?>
```

Our next task is to add the Flash Media Player to the JavaScript Gateway using the `addFlashObject` command. In Drupal, we can simply use the `drupal_add_js` routine to accomplish this, where we will provide the JavaScript code we wish to add to the header.

```
<?php
// Get the Flash application.
$sql = "SELECT * FROM {node} WHERE nid=%d";
```

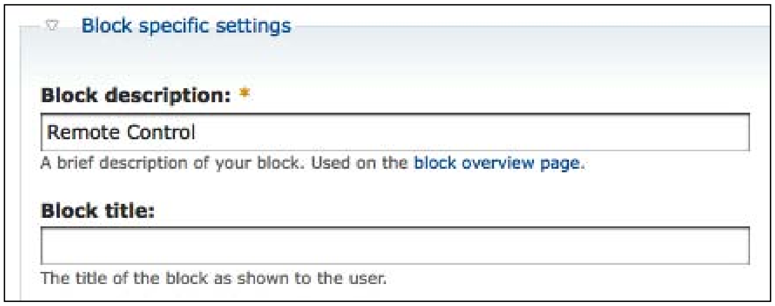
```
$flash = db_fetch_object(db_query($sql, 11));

// Load the flash node.
flashnode_load( $flash );

// Set the flashvars to the right node Id.
$flash->flashnode["flashvars"] = 'node=' . $node->nid;
$flash->flashnode["flashvars"] .= '&id=player';
$flash->flashnode["flashvars"] .= '&connect=remote';

// Add the Id of our player to the flashnode object.
$flash->flashnode["id"] = 'player';

// Add the player to the header.
drupal_add_js( 'addFlashObject ( \'player\' );', 'inline', 'header' );

// Show the Flash application.
print theme('flashnode', $flash->flashnode, FALSE);
?>
```

When we are done making these changes, we can then save our new video template and do the same thing for the audio node template. We are now ready to add our Remote Control, within the block regions of Drupal, to control the playback of the media connected to this node.

Adding the Remote Control

To add the Remote Control, we can either place the code within the same template that we just created, or we can use the Blocks section of our Drupal site. I prefer to use the Blocks section, since it really illustrates the separation between the two Flash applications. We can add a new block for our remote control by navigating to **Administer | Blocks**, where we will then click on the **Add Block** button.

For our remote control block, we can start off by giving our block a description of **Remote Control**, and then just leave the **Block title** blank.



Block description: *

Remote Control

A brief description of your block. Used on the block overview page.

Block title:

The title of the block as shown to the user.

For the **Block body**, we will start off with the same code as our video template, and then modify our FlashVars so that the `id` and `connect` variables are swapped. We can also remove the node FlashVar since the node information is not required for our `ControlBar` to control the media of a remote Flash application.

```
<?php
// Get the Flash application.
$sql = "SELECT * FROM {node} WHERE nid=%d";
$flash = db_fetch_object(db_query($sql, 11));
// Load the flash node.
flashnode_load( $flash );
// Set the flashvars.
$flash->flashnode["flashvars"] = 'id=remote';
$flash->flashnode["flashvars"] .= '&connect=player';
// Add the Id of our player to the flashnode object.
$flash->flashnode["id"] = 'remote';
// Add the player to the header.
drupal_add_js( 'addFlashObject ( \'remote\' );', 'inline', 'header' );
// Show the Flash application.
print theme('flashnode', $flash->flashnode, FALSE);
?>
```

We also need to make sure that we specify the new width and height of our remote control application. Since we really only wish to show the Play/Pause button, we can simply provide the dimensions of this button as follows:

```
// Set the flashvars to the right node Id.
$flash->flashnode["flashvars"] .= 'id=remote';
$flash->flashnode["flashvars"] .= '&connect=player';
// Set the width and height of our Flash application.
$flash->flashnode["width"] = 44;
$flash->flashnode["height"] = 33;
```

When we are done filling out our block body, we will now need to make sure that the **Input Format** for the block body is set to **PHP Code**, so that it will parse out our PHP code that we just provided.

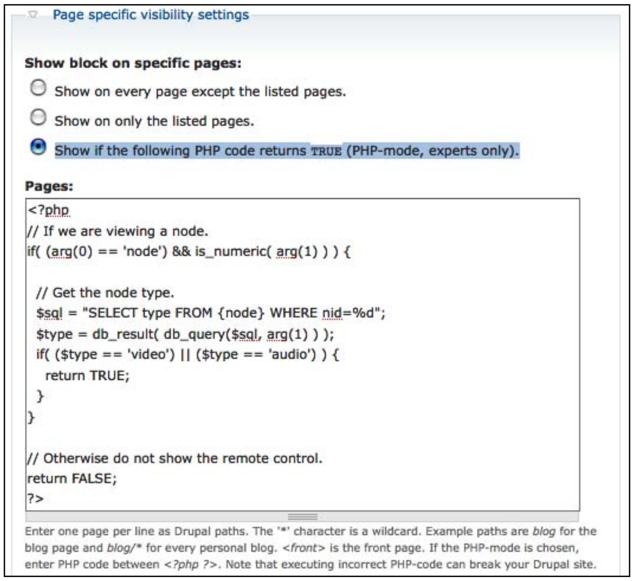
After we are done with this, our next task is to make sure that this block is only visible for the video and audio content types.

Adding block visibility for video and audio node types

Adding block visibility for specific node types can be a little tricky, since we do not have access to the visible node from within our block region. Since we do not have this information, we will need to use the path arguments to determine if we are viewing a node, and then get the node ID if we are. We can use the function `arg` within Drupal to determine the arguments for any page that is being viewed. Once we have the node ID, we can then perform a simple query to get the node type and then set our block as visible if the node type is either a video or audio node. We can do this by scrolling to **the Page Visibility Settings** and then placing the following code in the **Pages** text region.

```
<?php
// If we are viewing a node.
if( (arg(0) == 'node') && is_numeric( arg(1) ) ) {
  // Get the node type.
  $sql = "SELECT type FROM {node} WHERE nid=%d";
  $type = db_result( db_query($sql, arg(1) ) );
  if( ($type == 'video') || ($type == 'audio') ) {
    return TRUE;
  }
}
// Otherwise do not show the remote control.
return FALSE;
?>
```

We then need to make sure that we check the PHP radio button, so that it uses our code to display our block or not.



Page specific visibility settings

Show block on specific pages:

○ Show on every page except the listed pages.

○ Show on only the listed pages.

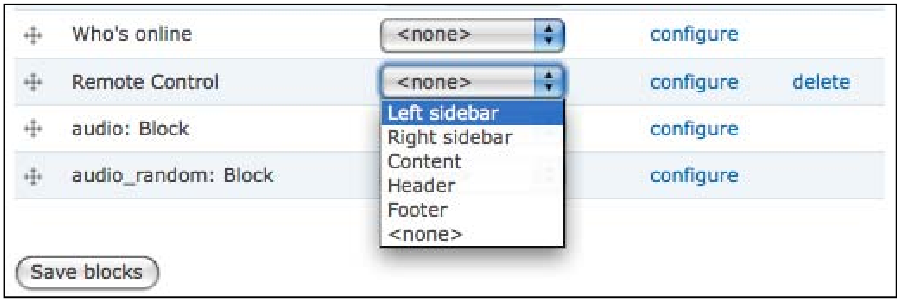
● Show if the following PHP code returns TRUE (PHP-mode, experts only).

Pages:

```
<?php
// If we are viewing a node.
if( (arg(0) == 'node') && is_numeric( arg(1) ) ) {

 // Get the node type.
 $sql = "SELECT type FROM {node} WHERE nid=%d";
 $type = db_result( db_query($sql, arg(1) ) );
 if( ($type == 'video') || ($type == 'audio') ) {
   return TRUE;
 }
}

// Otherwise do not show the remote control.
return FALSE;
?>
```

Enter one page per line as Drupal paths. The '*' character is a wildcard. Example paths are *blog* for the blog page and *blog/** for every personal blog. *<front>* is the front page. If the PHP-mode is chosen, enter PHP code between *<?php ?>*. Note that executing incorrect PHP-code can break your Drupal site.

Once we save our new Block, we should then be taken to the list of all available blocks, where we will see our new **Remote Control** block in the list of disabled blocks. We can enable it by clicking on the **Region** drop-down box and selecting **Left Sidebar** as follows:



After we do this, it should then add the **Remote Control** to the left region, where we can then click on the **Save blocks** button at the bottom of the page to save our changes.

After we save the blocks, we can then visit any of our media nodes, where we should then see our new remote control application show up in the left sidebar. But you may have also noticed that our Play/Pause button does not show up on either the remote control or the media player. The reason this is occurring is because the FlashNode default template does not allow us to provide the ID for our objects that are embedded within the page. Since the object IDs are required to make the remote connections between the remote control and the media player, each player is stuck waiting for its corresponding player to register. Although this may seem like quite a roadblock, we can still fix this fairly easily by overriding the default FlashNode template so that we can provide our own custom IDs for our Flash objects.

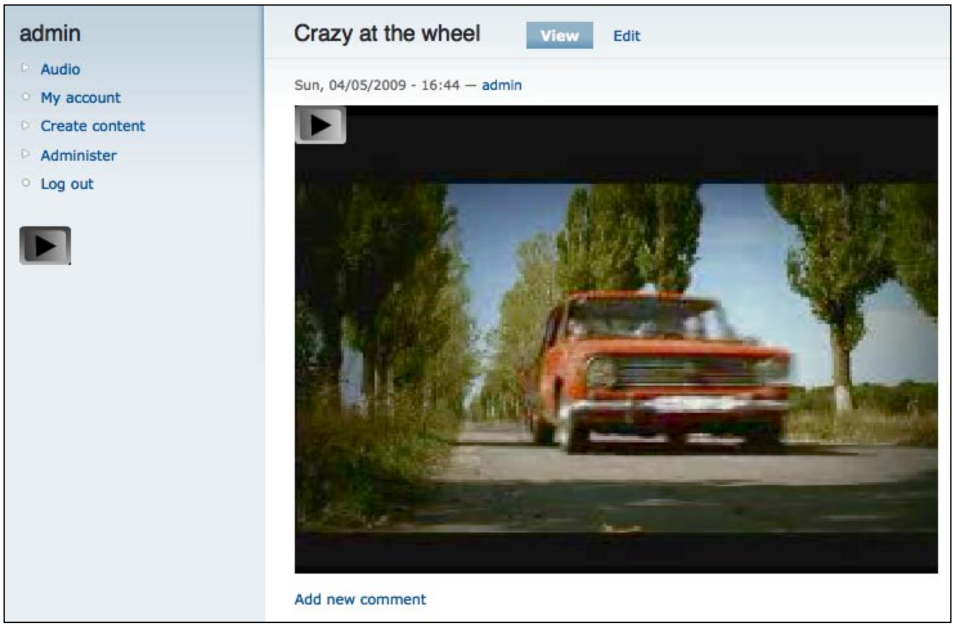
Creating a FlashNode template

In order to override the FlashNode template, we will first need to locate the `template.php` file within our Drupal theme. If we are sticking with the defaults, then this file can be found within the Garland theme located in the `themes` folder. When we open up this file, we can then override any theme call, made by any module, and replace it with our own custom functionality. This is a very handy trick if you ever wish to override any visible component within Drupal with your own customized version of that component.

For our case, we will need to override the `theme_flashnode_markup` function since it is responsible for drawing the object code for each Flash application that we submit using the FlashNode module. Within the `template.php` file, we can then override this function by declaring a new function that will override the `theme_flashnode_markup` function by replacing the `theme` with `phptemplate`. We can then provide our very own object code and use the variables passed to our FlashNode theme to populate a new theme for any FlashNode object submitted to our site. So, at the bottom of our `template.php` file, we can place the following code to override the FlashNode theme.

```
/**
 * Override the Flash node template
 */
function phptemplate_flashnode_markup( $flashnode, $options = array()
) {
  // Create path to the swf file
  $filepath = file_create_url($flashnode['filepath']);
  // Create our Flash object.
  $output = '<object ';
  $output .= 'width="'. $flashnode['width'] .'" ';
  $output .= 'height="'. $flashnode['height'] .'" ';
  $output .= 'id="'. $flashnode['id'] .'">';
  $output .= '<param name="movie" value="'. $filepath .'" />';
  $output .= '<param name="flashvars" value="'.
  $flashnode['flashvars'] .'" />';
  $output .= '<embed src="'. $filepath .'" ';
  $output .= 'width="'. $flashnode['width'] .'" ';
  $output .= 'height="'. $flashnode['height'] .'" ';
  $output .= 'flashvars="'. $flashnode['flashvars'] .'" ';
  $output .= 'name="'. $flashnode['id'] .'" />';
  $output .= '</object>';
  return $output;
}
```

After we have done this, we should now be able to navigate to any audio or video node, where we will be met with a media player with a completely separate remote control!



Summary

We covered a lot of ground in this chapter, by picking up where we left off in the previous chapter, and by creating the necessary components required to implement the hybrid approach. This was achieved by walking through the following steps:

1. We first abstracted out all component functionality into separate classes, where the `ControlBar` was a separate class from the `MediaPlayer`.
2. We built a static communication gateway between those classes. For our example, this was the `MediaGateway` component.
3. We built a JavaScript gateway to connect two separate Flash applications.
4. We synchronized each Flash application using the JavaScript gateway.

In the next chapter, we will continue our exploration into other ways to extract data from Drupal by creating lists of content using the very popular Views module.

9
Flash with Drupal Views

Up until this point, we have only been dealing with single pieces of content within Drupal called nodes. Within each node, we have learned how to manage any form of data that Drupal can throw our way. Our next step is to learn how to handle lists of nodes within Flash using the extremely popular Views module. Using the Views module within Flash can be extremely powerful, since it gives us complete control over how we would like to filter our content within our Flash application and then deliver that content in list form. In this chapter we will explore how to build a Flash application that displays lists of nodes, using the Views module, by creating a playlist for the media player that we have been building throughout this book. We will accomplish this by taking the following steps:

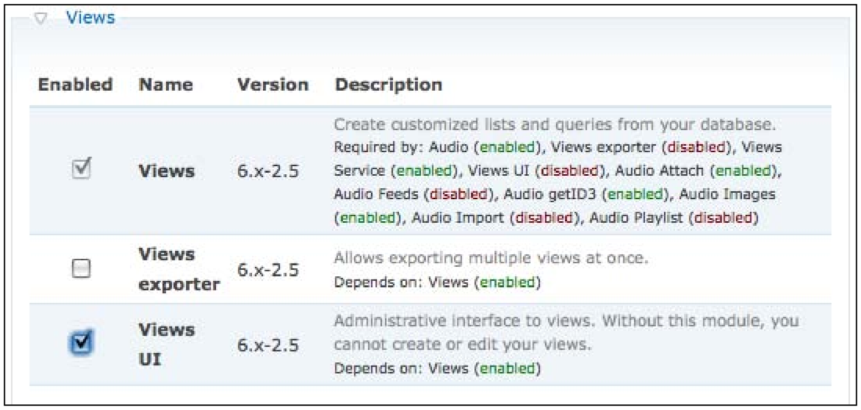
- Using the Drupal Views module
- Using the Views Service
- Building a Flash Playlist using Drupal
- Creating a node teaser
- Building a `ListView` class

Using the Drupal Views module

The Views module is one of the most popular, contributed modules for Drupal, and is so for a very good reason. It is a fantastic module that allows Drupal site administrators to create lists of nodes using custom filter criteria. The best way to describe how this module is used is to give the example of a recipe web site, which has many Recipe nodes. The first thing we can expect to see, when visiting any recipe web site, will be a list of recipes that link to their individual recipe nodes. The Views module gives the Drupal administrator the ability to create these lists of recipes that match any type of filter criteria imaginable. This concept can then be easily applied to our media player, where a view could be used to list all of our Drupal media nodes within our Flash application. But to really understand how this module works, we will first install this module in our site, where we will then experience first hand the power of this incredible module.

Views: Installation and Configuration

We will start this section by first downloading the Drupal Views module by visiting `http://www.drupal.org/project/views`, and then placing the contents of this package within your site's `modules` folder. Once you have the views package on your server, we can then navigate the **Modules** section of our Drupal Administrator. Scrolling down to the Views section, we can enable the Views module by checking the **Views** and **Views UI** as shown:



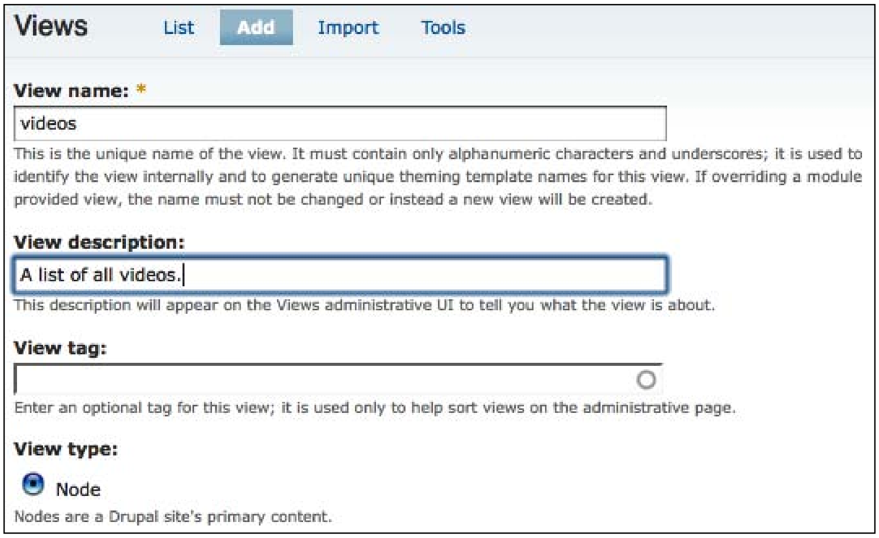
Enabled	Name	Version	Description
☑	Views	6.x-2.5	Create customized lists and queries from your database. Required by: Audio (enabled), Views exporter (disabled), Views Service (enabled), Views UI (disabled), Audio Attach (enabled), Audio Feeds (disabled), Audio getID3 (enabled), Audio Images (enabled), Audio Import (disabled), Audio Playlist (disabled)
□	Views exporter	6.x-2.5	Allows exporting multiple views at once. Depends on: Views (enabled)
☑	Views UI	6.x-2.5	Administrative interface to views. Without this module, you cannot create or edit your views. Depends on: Views (enabled)

Now, we just need to click on the **Save Configuration** button at the bottom of the page to commit this change. After we have enabled the Views module, we can set it up to show a listing of all media nodes in our Drupal system.

Setting up a view

Our next task is to set up a view to display a list of nodes in our Drupal system. Since we have already set up the Audio and Video node types, we can now easily add a view to filter all of our Drupal content based on the node type of each piece of content. To keep this chapter as simple as possible, we will only concentrate on the Video node type when building and showing our view within Flash. However, it is important to note that these same steps can also be applied for the Audio node type that we created in a previous chapter. So, let's begin by first creating a new view to show all of our Video nodes in our Drupal web site by navigating to the **Administer | Views** section.

Once we are in the Views administrator section, we will create a new view by clicking on the **Add** button at the top of the screen. This will then bring up a new page, where it asks for us to provide a view name and other information regarding the view that we are creating. We can now provide the following information, and then click on the **Next** button at the bottom of the page when we are done.



View name: *

videos

This is the unique name of the view. It must contain only alphanumeric characters and underscores; it is used to identify the view internally and to generate unique theming template names for this view. If overriding a module provided view, the name must not be changed or instead a new view will be created.

View description:

A list of all videos.

This description will appear on the Views administrative UI to tell you what the view is about.

View tag:

Enter an optional tag for this view; it is used only to help sort views on the administrative page.

View type:

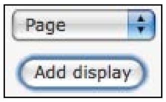
● Node

Nodes are a Drupal site's primary content.

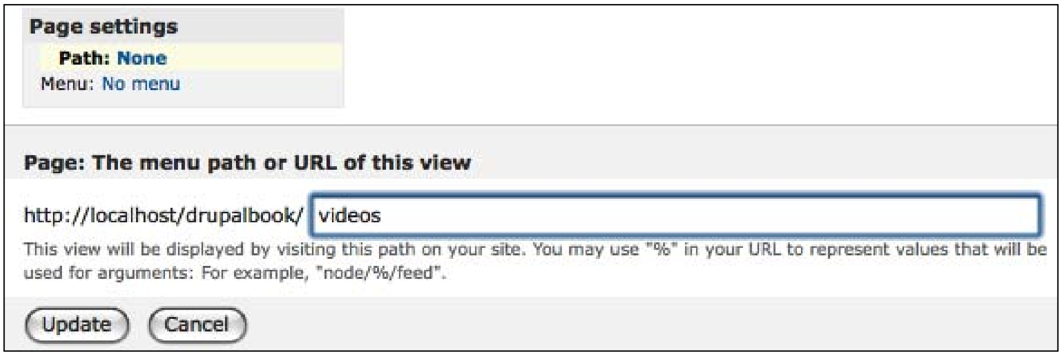
After we click on the **Next** button, we should then see a new page where we can set up our new view. The views module gives us complete control over what information we want to show in our view as well as how we want to show it. Although providing many different parameters to configure our view is extremely powerful, this may also seem very intimidating to a person who is not familiar with this module. To help make this section as simple as possible, we will walk through the process of setting up a view by tackling each piece individually, starting with creating a new page view.

Creating a new page view

In this section, we will create a **page view**, which is essentially a new page that will show the result of our filtered content. This is a good first step so that we can have the option to view our content lists using either HTML or Flash (which we will create later). In order to create a page view, our first task will be to click on the **Add display** button with the **Page** selected in the drop-down box.



Now that we have created a page, our next step will be to provide a path so that we can view the page that we created. We can do this by clicking on the **None** link in the **Path Settings**. This will then bring up a new section, where we can provide a new path, which we will call **videos**. After that, we can then click on the **Update** button to commit the change.



Page settings

Path: None

Menu: No menu

Page: The menu path or URL of this view

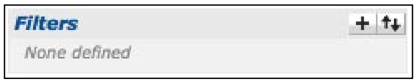
http://localhost/drupalbook/ videos

This view will be displayed by visiting this path on your site. You may use "%" in your URL to represent values that will be used for arguments: For example, "node/%/feed".

Now that we have a page view defined, our next task is to add some fields that will be shown when the user navigates to this path in their browser.

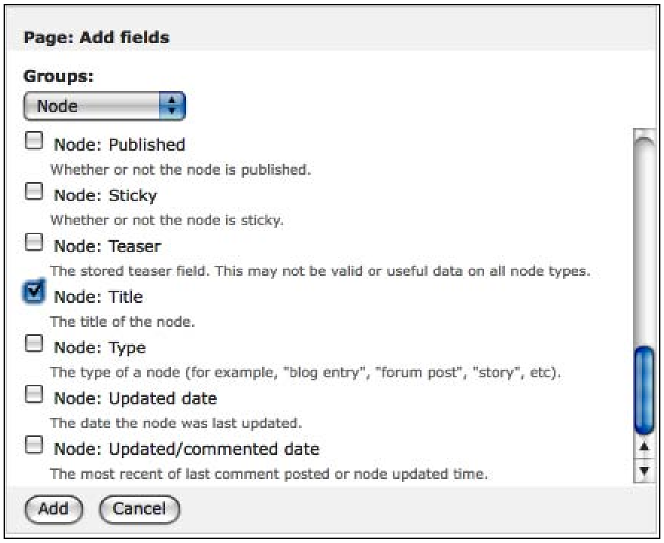
Adding fields to a view

The fields for a view are used to display specific node information when the user navigates to the path that we defined in the previous section. We can set up our fields by clicking on the + symbol, next to the **Fields** section, of our view setup screen.



None defined

This should bring up a new section of the screen that will allow us to select which fields we would like to display in our view. Since we are really only concerned with node information, we can filter these fields by first selecting **Node** in the **Groups** drop-down box. We can then select the checkbox next to the **Node : Title** and click on the **Add** button.



Page: Add fields

Groups:

Node

□ Node: Published
Whether or not the node is published.

□ Node: Sticky
Whether or not the node is sticky.

□ Node: Teaser
The stored teaser field. This may not be valid or useful data on all node types.

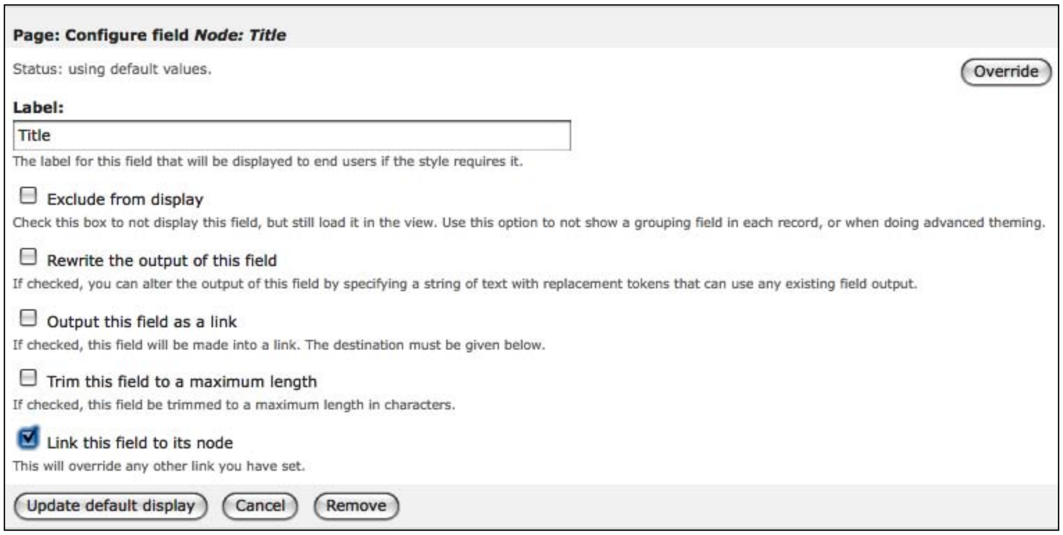
☑ Node: Title
The title of the node.

□ Node: Type
The type of a node (for example, "blog entry", "forum post", "story", etc).

□ Node: Updated date
The date the node was last updated.

□ Node: Updated/commented date
The most recent of last comment posted or node updated time.

After clicking on the **Add** button, we will be given another section that will allow us to provide specific configurations for the field that we just added. Since we will want each title to link to the node, we will click on the checkbox where it says **Link this field to its node**, and then click on the **Update default display** button.



Page: Configure field *Node: Title*

Status: using default values.

Label:

Title

The label for this field that will be displayed to end users if the style requires it.

□ Exclude from display
Check this box to not display this field, but still load it in the view. Use this option to not show a grouping field in each record, or when doing advanced theming.

□ Rewrite the output of this field
If checked, you can alter the output of this field by specifying a string of text with replacement tokens that can use any existing field output.

□ Output this field as a link
If checked, this field will be made into a link. The destination must be given below.

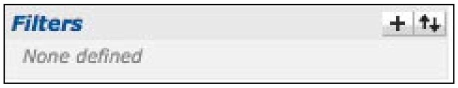
□ Trim this field to a maximum length
If checked, this field be trimmed to a maximum length in characters.

☑ Link this field to its node
This will override any other link you have set.

Our last step in setting up our view will be to filter the content by using the Filter section of our view configuration.

Adding a Filter to our view

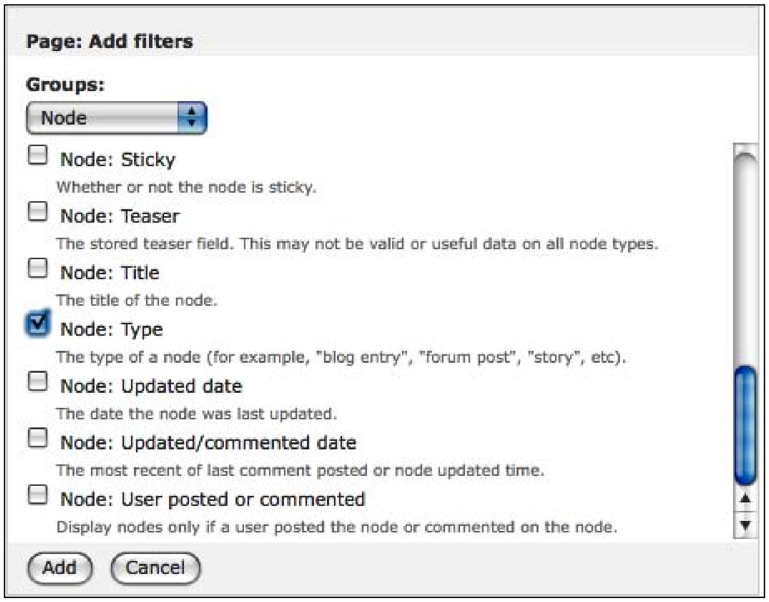
For this last section, we want to tell our view that we are only concerned with Video nodes that are published. From that statement, we can determine that we will need some mechanism to filter our view based on the node type of that content. This can be accomplished by clicking on the + next to the **Filters** section.



Filters

None defined

This will then bring up a new section, where we can provide the node type as a filter. We can do this by first clicking on the **Groups** drop-down box and selecting the **Node** group. We will then click on the checkbox next to **Node : Type**, and click on the **Add** button.



Page: Add filters

Groups: Node

□ Node: Sticky
Whether or not the node is sticky.

□ Node: Teaser
The stored teaser field. This may not be valid or useful data on all node types.

□ Node: Title
The title of the node.

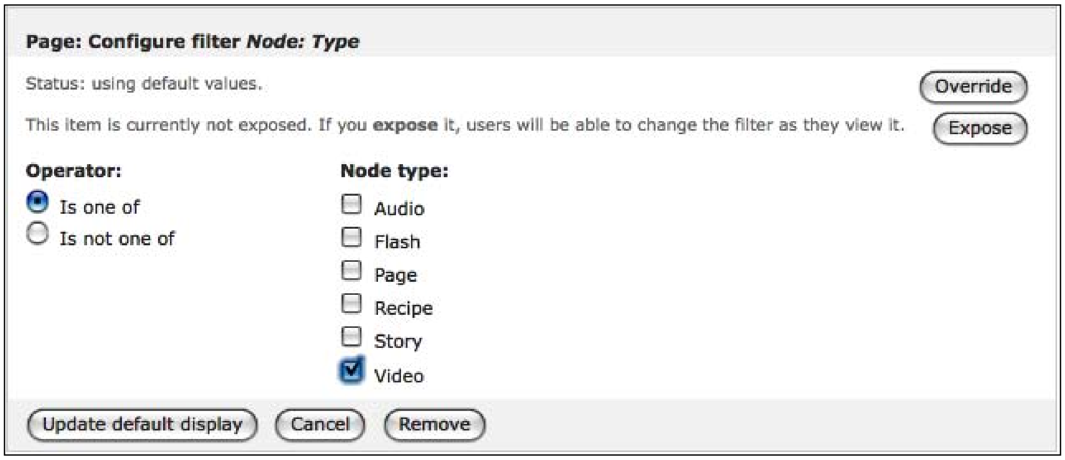
☑ Node: Type
The type of a node (for example, "blog entry", "forum post", "story", etc).

□ Node: Updated date
The date the node was last updated.

□ Node: Updated/commented date
The most recent of last comment posted or node updated time.

□ Node: User posted or commented
Display nodes only if a user posted the node or commented on the node.

This will then bring up a new section, where we can select which node types we would like to use to filter our content. We will then select the **Video** node type and click on the **Update default display** button to add it to our filter list.



Page: Configure filter *Node: Type*

Status: using default values.

This item is currently not exposed. If you **expose** it, users will be able to change the filter as they view it.

Operator:

- ● Is one of
- ○ Is not one of

Node type:

- □ Audio
- □ Flash
- □ Page
- □ Recipe
- □ Story
- ☑ Video

Now that we have completed setting up our view, it is extremely important to save all of our changes by clicking on the **Save** button at the bottom of the page. After we have saved our view, we can visit this view to see a listing of all videos on our Drupal site by navigating to `http://localhost/drupal6/videos` in our browser. If you do not have anything showing, then this would probably be a good time to add 5-10 videos on your site to help test the Views integration with Flash. For a reference on how to add new videos to our site, we can easily refer back to Chapter 6, where we built a Video player and added a test video node to Drupal.

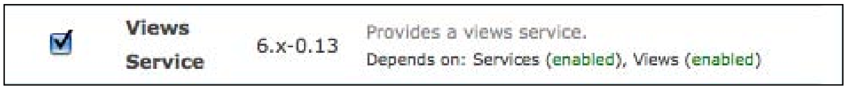
After we have several videos on our site, we will explore the Views Service module, which provides us with a mechanism for extracting view information within Flash.

Using the Views Service

The Views Service is a Service Module that utilizes the Views module to expose lists of nodes to outside applications. Now that we have the Views module enabled and configured, we can use this service to provide the lists of nodes that we will use to construct our media playlist within Flash. In order to effectively use this module, it is important to walk through a few steps to make sure that it is installed and configured correctly.

Step 1: Install the Views Service

We will install the Views service the same way that we installed all other modules within Drupal. Fortunately, however, we will not need to download a separate module from Drupal since the Views service module comes pre-packaged with the Services module download. Because of this, we can simply navigate to our **Modules** section in the Drupal administrator and scroll down to the Services section and enable the **Views Service**.

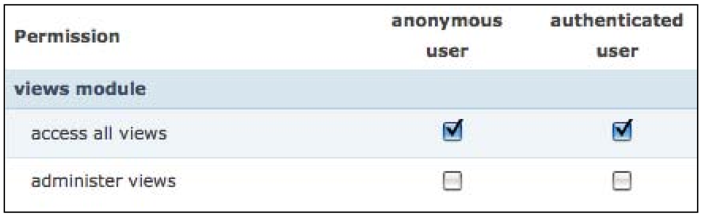


☑	**Views Service**	6.x-0.13	Provides a views service. Depends on: Services (enabled), Views (enabled)

Now that the Views service is enabled, our next step is to ensure that the user permissions are set correctly to allow us to extract view information.

Step 2: Configure user permissions

In order to use the Views service to extract view data, we will first need to navigate to the User Permissions section by going to **Administer | Permissions**. Here, we will need to enable the **access all views** permission within the **Views** section for all user groups.

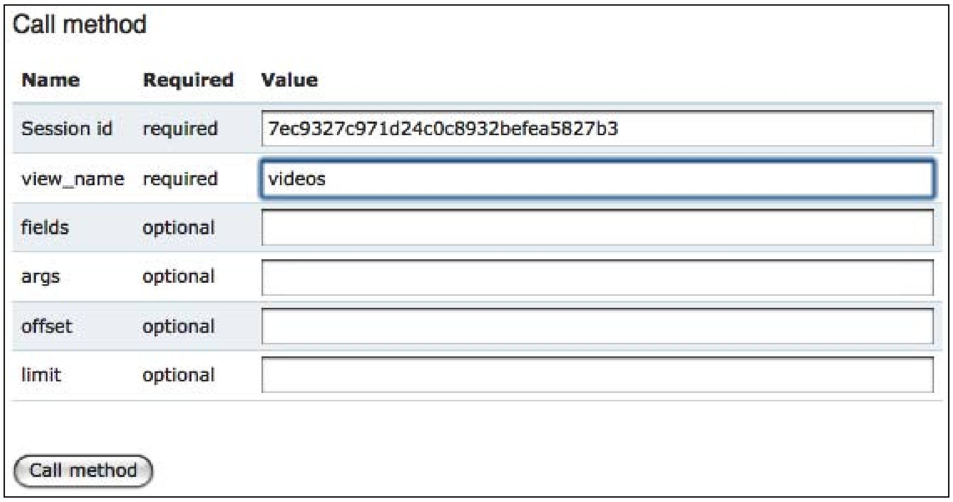


Permission	anonymous user	authenticated user
views module		
access all views	☑	☑
administer views	□	□

> If you have any views that you do not wish to be made public, you will need to modify this permission to only enable the authorized user groups. However, by doing this, you will keep the remote debugging from functioning since the remote debugging basically acts as an anonymous user. If your web site contains sensitive data, within your view, then I highly recommend enabling these permissions for debugging purposes only, but then unchecking them when the Flash application is placed on your server. By having the Flash application reside on the server, it will also use the same session ID associated with a certain user group and therefore, make these permissions function as intended.

Step 3: Verify it works

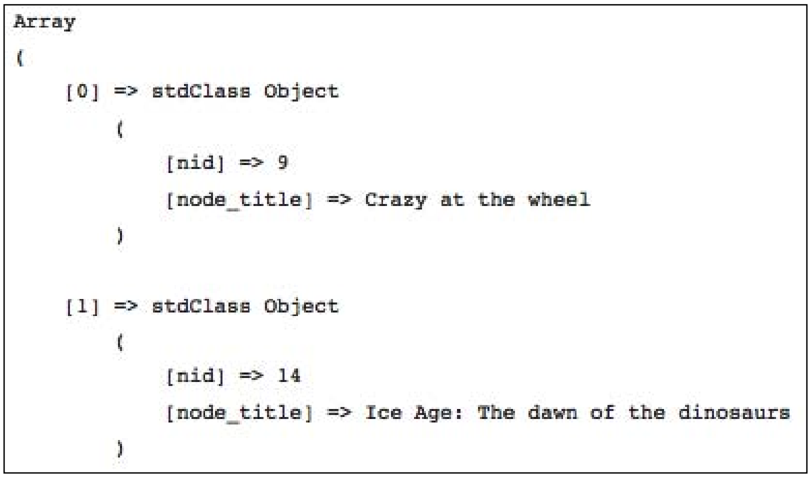
Our final step is to make sure that the Views service that we just implemented does indeed serve a list of nodes. We can test this out by navigating to the Services Administrator at **Administer | Services**. Once we are there, we should see a new service in the Services list called **views.get**. By clicking on this link, we can now use the Services Administrator to test and make sure that the Views Service really works as expected. Once we click on the **views.get** link, we should see a page that shows the function arguments required to execute this function from a remote location. It also gives us the ability to provide our own data, call the method as if any external application was making the call, and then observe the response. To do this, we will just provide the only required argument for this method, which is the name of the view that we would like to call. We can test out this service by placing the **videos** view name in this text box, and then clicking the **Call method** button. Also, keep in mind that the Session ID is a randomly generated value, so we will not need to change that value from what is provided.



Call method

Name	Required	Value
Session id	required	7ec9327c971d24c0c8932befea5827b3
view_name	required	videos
fields	optional	
args	optional	
offset	optional	
limit	optional	

After we call this method, we should see something similar to the following:



```
Array
(
    [0] => stdClass Object
        (
            [nid] => 9
            [node_title] => Crazy at the wheel
        )

    [1] => stdClass Object
        (
            [nid] => 14
            [node_title] => Ice Age: The dawn of the dinosaurs
        )
```

Our next task is to take this data and incorporate it into a playlist for our Flash media player.

Building a Flash Playlist using Drupal

Before we begin this section, we will first need to copy the `chapter8` folder, and then paste it as a new folder called `chapter9` so that we can keep all changes made to our media player separate from the previous chapter. After we have done this, we will start out by opening up the `main.as` file, where we will add some functionality to load a Drupal view. The first task will be to create a new function called `loadView` that we will use to encapsulate the responder and service call to Drupal. We can place this function below our `loadNode` function as follows:

```
// Loads a Drupal node.
function loadNode( nid:Number )
{
   // Set up our responder with the callbacks.
   var nodeResponse:Responder = new Responder( onNodeLoad, onError);
   // Call Drupal to get the node.
   drupal.call( "node.get", nodeResponse, sessionId, nid );
}
// Loads a Drupal view.
function loadView( _viewName:String )
{
   // Set up the responder with the callbacks.
```

```
      var viewResponse:Responder = new Responder( onViewLoad,onError );
      // Call Drupal to load the view.
      drupal.call( "views.get", viewResponse, sessionId, _viewName );
   }
```

Our next task will be to create the handler function that will handle the contents of the view that we get from Drupal. We can do this by simply adding the following function after the onNodeLoad function.

```
   // Called when Drupal returns with our node information.
   function onNodeLoad( node:Object )
   {
       ...
      ...
   }
   // Called when a view gets loaded.
   function onViewLoad( _view:Object )
   {
      trace( "View loaded" );
   }
```

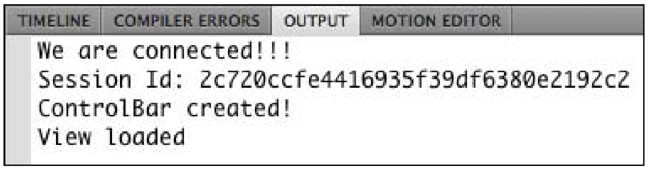
After we have done this, we can create our global view name variable that will be used to handle the view name passed to our player using FlashVars. We can do the same thing that we did for the nodeId variable by declaring this new global variable as follows. And since we will want to test this new view, we can temporarily hard code the value of this variable with the name of the view that we created in the previous section.

```
   // Declare our variables
   var baseURL:String = "http://localhost/drupal6";
   var gateway:String = baseURL + "/services/amfphp"
   var sessionId:String = "";
   var nodeId:Number = root.loaderInfo.parameters.node;
   var viewName:String = "videos";
   var id:String = root.loaderInfo.parameters.id;
   var connect:String = root.loaderInfo.parameters.connect;
```

Now that we have our view variable in place, we can test to make sure that this works by modifying the onConnect function, so that it will load the view if a name was provided.

```
   // Called when Drupal returns with a successful connection.
   function onConnect( result:Object )
   {
      // Set our sessionId variable.
      sessionId = result.sessid;
      trace("We are connected!!!");
      trace("Session Id: " + sessionId);
      // If the node Id is valid...
      if( nodeId ) {
         // Load our node.
         loadNode( nodeId );
      }
      else {
         // Add a control bar.
         addControlBar();
      }
      // If they provided a view name.
      if( viewName ) {
         // Load the view.
         loadView( viewName );
      }
   }
```

We should now be able to open up our `chapter9.fla` project, and then run our media player in Flash to see that the view gets loaded by looking at the output panel in our Flash application.



```
We are connected!!!
Session Id: 2c720ccfe4416935f39df6380e2192c2
ControlBar created!
View loaded
```

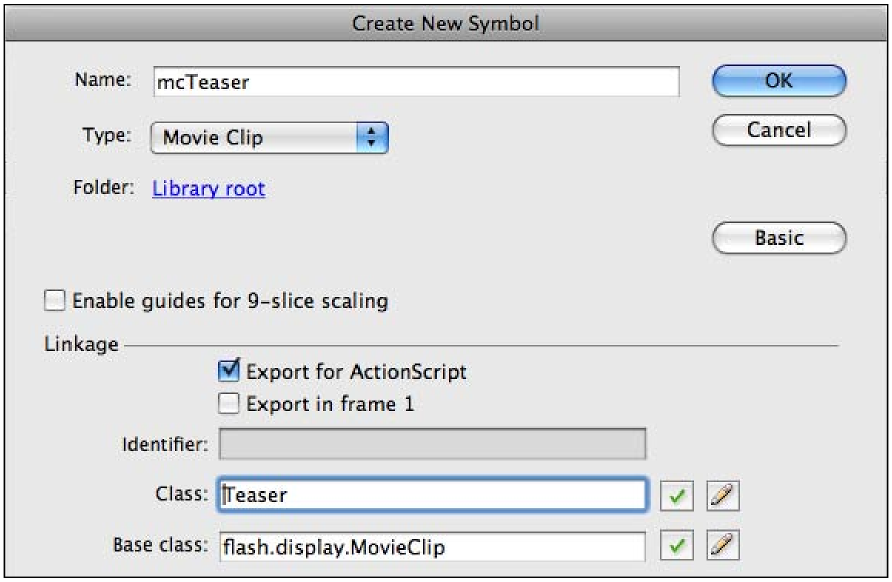
Our next task is to iterate through all of our nodes, within the view, and parse out the data that we need. We can do this by adding the following code to our `onViewLoad` function.

```
   // Called when a view gets loaded.
   function onViewLoad( _view:Object )
   {
      // Iterate through all of the nodes.
      for each( var node:Object in _view )
      {
         trace( node.nid );
      }
   }
```

If we were to run our application again, we would see all of the nodes in our view print out in the debug window. We are now ready to construct a new movie clip along with a new `ActionScript` class that we will use to display each teaser in our view.

Creating a node teaser

Our first task in this section will be to create a new Teaser movie clip that will be used to contain all the node information provided with the Views module. We can start this process by first selecting the menu item **Insert | New Symbol** in our Flash IDE and creating a new movie clip called **mcTeaser**. We will also want to make sure that we link that new movie clip to an ActionScript class (which we will create later) called **Teaser**.



Name: mcTeaser

Type: Movie Clip

Folder: Library root

☐ Enable guides for 9-slice scaling

Linkage

☑ Export for ActionScript

☐ Export in frame 1

Identifier:

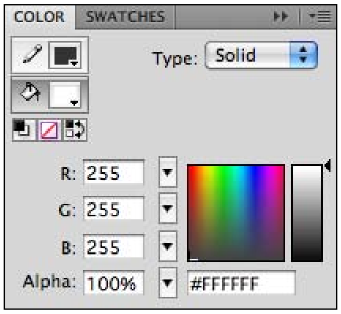
Class: Teaser

Base class: flash.display.MovieClip

After we accept the warning for creating a class that is not yet found, we should be given a blank stage, where we can then build our teaser the way we want it to look. At this point, we will want to step back and decide how we want our teasers to behave when the mouse moves over them, clicks them, and selects them. Given most typical list applications, we can determine that we will need a **hover**, **selected**, and **normal** state for our teasers. For each one of these states, we can simply just change the color of the background of the teaser to make these states noticeable. So, we will first focus our attention on creating a background that allows for three different states.

Creating a teaser background

To create a teaser background, we will first want to add a rectangle region that will be used to indicate the **normal** state of our background by first clicking on the ▭ symbol in our toolbar. This will require us to decide what color we would like our background to be. We can do this by opening up the **Color** window, which should look similar to the following:

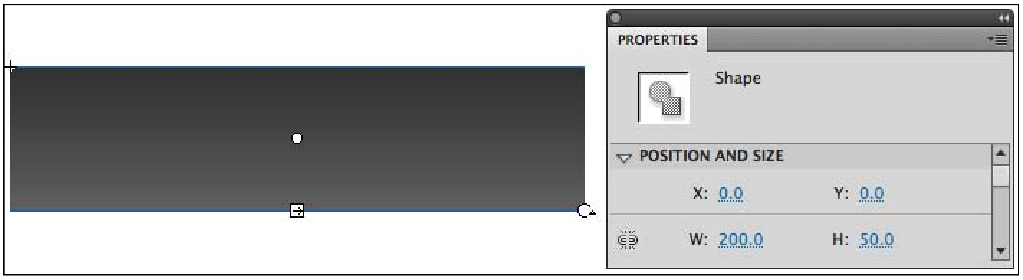


Type: Solid

R: 255
G: 255
B: 255
Alpha: 100%
#FFFFFF

Typically, we would not want any border around a teaser, so we can disable the border of the rectangle by clicking on the ✎■ symbol within the **Color** window, and then selecting ⧄ from the color toolbar. For the fill color, we will first click on the ◈■ symbol, and then select **Linear** for the fill type as follows. Type: Linear This will give us a linear gradient fill that will give our teasers some depth.

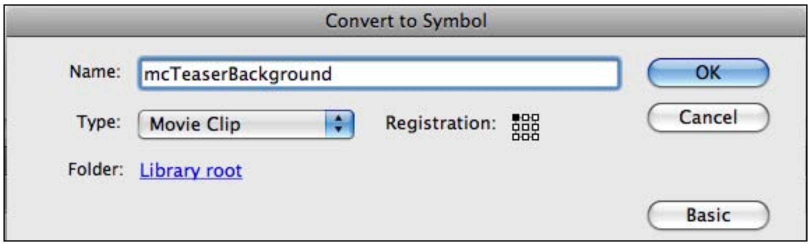
With the **Linear** fill type selected, we can now choose our colors for each gradient pole using the ▬ tool. Since we have already done this in previous chapters, we can choose a color of **0x333333** at one pole, and **0x666666** at the other. We can now draw a 200 pixel wide and 50 pixel tall rectangle, and then move the orientation of the gradient so that it looks like the following:



Shape

X: 0.0
Y: 0.0
W: 200.0
H: 50.0

Once this background has been made, we will need to turn it into a movie clip by selecting the rectangle, and then clicking on **Modify | Convert to Symbol** and give it a name of **mcTeaserBackground**.



Convert to Symbol

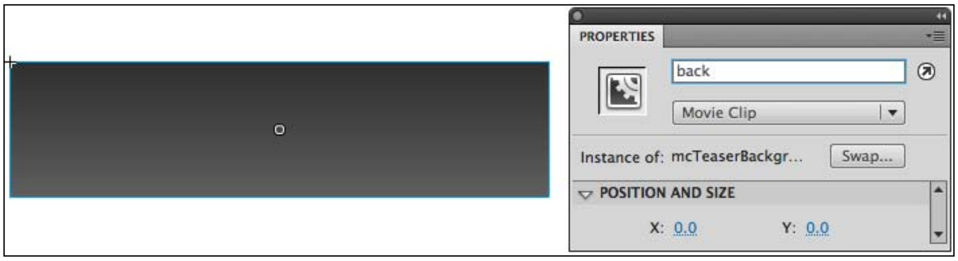
Name: mcTeaserBackground

Type: Movie Clip

Registration:

Folder: Library root

Once our movie clip has been added, we need to click on the new movie clip and give it an instance name of **back**.



PROPERTIES

back

Movie Clip

Instance of: mcTeaserBackgr...

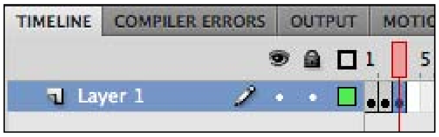
POSITION AND SIZE

X: 0.0 Y: 0.0

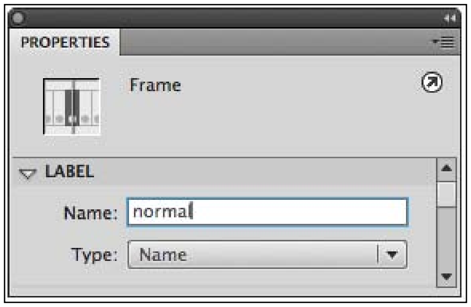
Now that our background has an instance name, we can enter this movie clip, where we will add three different states for our background.

Using the timeline to add different teaser states

Now that we are inside of our background movie clip, we can utilize the timeline to add our three background states of **normal**, **hover**, and **selected**. We will start this out by first adding two new frames to our timeline by clicking on the Timeline window, and then pressing the *F6* key twice while the cursor is in the first frame of the first layer. When we are done, we should have something that looks like the following:



Our next task is to label each of these frames so that they can be referenced within ActionScript. To label a frame, all we have to do is click on the frame that we would like to label, and then in the **Properties** tab, we can provide a label for that frame. For the first frame, we will call it **normal**.



Name: normal

Type: Name

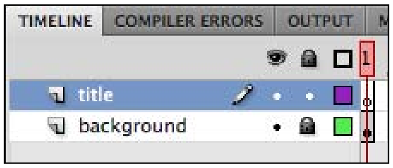
We will now do the same thing for the other two key frames, but this time we'll give them the labels of **hover** and **selected** respectively. Our last and final task is to change the background for the hover and selected frames so that they look different from the normal frame. We can do this by simply changing the fill gradient colors for each of these frames. For the hover frame, we will change the gradients to 0x666666-0x999999, and for the selected frame, we will change the gradients to 0x000000-0x333333. When we are done, we should then have a new teaser background that will handle all three of the teaser states.

We can now exit this movie clip, where we will set up the title text for this teaser.

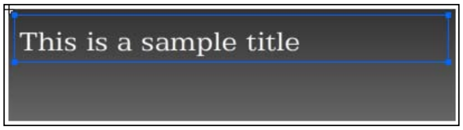
Adding a title to the teaser

The first thing that we will do in this section is make sure that we are back in the **mcTeaser** movie clip by looking at the breadcrumb at the top of the stage.

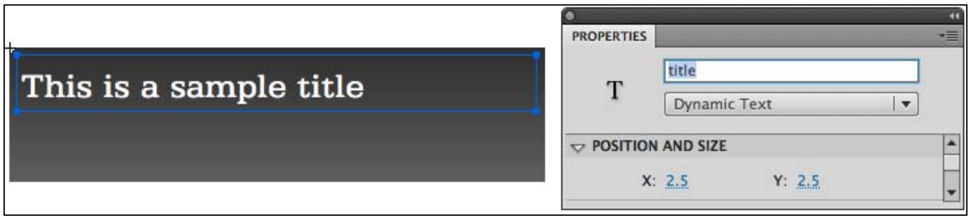
Once we are within our **mcTeaser** movie clip, we will first want to separate out the **background** from the **title** using the layers. So, we will start by labeling the layer that contains our **background**, will then add a new layer that we will call **title**, and will finally then lock the background layer since we are done with it.



With our **title** layer selected, we now need to add a new Dynamic text field using the T symbol from the toolbar, and then draw a text region at approximately the same size as the background, and type some default text using a white color for the characters.



Once we have our new **TextField** in place, we will then need to give our new **TextField** an instance name using the properties windows. We will call our Dynamic TextField **title** since we will use it to show the title of each node in our view list.



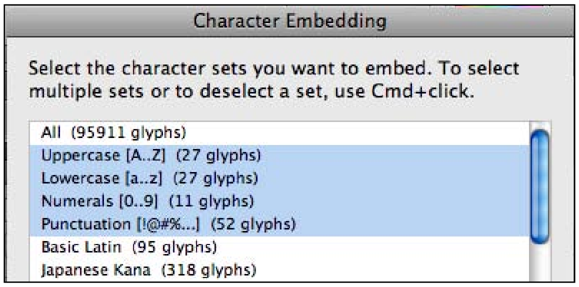
title

Dynamic Text

POSITION AND SIZE

X: 2.5 Y: 2.5

Finally, we will need to make sure that the characters that we are using for this Dynamic TextBox will be embedded within the application. We can do this by clicking on the button within the Properties section called **Character Embedding**. When the character window shows up, we can select the following character sets to embed into our Flash application.



Character Embedding

Select the character sets you want to embed. To select multiple sets or to deselect a set, use Cmd+click.

All (95911 glyphs)
Uppercase [A..Z] (27 glyphs)
Lowercase [a..z] (27 glyphs)
Numerals [0..9] (11 glyphs)
Punctuation [!@#%...] (52 glyphs)
Basic Latin (95 glyphs)
Japanese Kana (318 glyphs)

After you click on **OK**, we are now done setting up our teaser movie clip. Our next task is to write the ActionScript class that will be used to give our teaser some functionality.

Creating a Teaser class

Since we linked our teaser movie clip to a **Teaser** class within the movie clip properties, we can now build a new class that will govern the behavior of the movie clip elements that we just created. We will start this by creating a `Teaser.as` file, within our `chapter9` directory, and then writing a stub class called **Teaser**. Our stub class should look something similar to the following:

```
package
{
   // Import all dependencies
   import flash.events.Event;
   import flash.display.MovieClip;
   import flash.text.TextField;
   // Declare our class
   public class Teaser extends MovieClip
   {
      // Constructor function.
      public function Teaser()
      {
         addEventListener( Event.ADDED_TO_STAGE, onAdded );
      }
      // Called when this object has been added to the stage.
      private function onAdded( event:Event )
      {
      }
      public var back:MovieClip;
      public var title:TextField;
   }
}
```

Now that we have our stub class, we can start out by setting up the correct behavior for our background region. Basically, what we are after is initializing our background to the **normal** state when the teaser is added to the stage, and then switching the background to use the **hover** state when the mouse hovers over the teaser. We will also need to make sure that the background goes back to the **normal** state when the mouse moves out of the teaser region. All of these changes will look like the following:

```
package
{
   // Import all dependencies
   import flash.events.Event;
   import flash.events.MouseEvent;
   import flash.display.MovieClip;
   import flash.text.TextField;
   // Declare our class
```

```
    public class Teaser extends MovieClip
    {
       // Constructor function.
       public function Teaser()
       {
          addEventListener( Event.ADDED_TO_STAGE, onAdded );
          addEventListener( MouseEvent.MOUSE_OVER, setHover );
          addEventListener( MouseEvent.MOUSE_OUT, setNormal );
          addEventListener( MouseEvent.MOUSE_DOWN, setNormal );
       }
       // Called when this object has been added to the stage.
       private function onAdded( event:Event )
       {
          // Declare this teaser as a button.
          buttonMode = true;
          mouseChildren = false;
          // Go to the normal state.
          back.gotoAndStop("normal");
       }
       // Called when the teaser is hovered over.
       private function setHover( e:MouseEvent )
       {
          back.gotoAndStop("hover");
       }
       // Called when the mouse moves out of the teaser.
       private function setNormal( e:MouseEvent )
       {
          back.gotoAndStop("normal");
       }
       public var back:MovieClip;
       public var title:TextField;
    }
 }
```

Our next task is to keep track of the **selected** state. Basically, how this will work is we will create a public function that will be called to set our teaser as selected or not. Also, we do not want to trigger the **normal** and **hover** states when this variable is set to true. The reason for this is we do not want our teaser to appear as if it has reset back to a **normal** state when it is really still **selected**. Following are the changes needed to incorporate the selected state of our teaser:

```
package
{
   // Import all dependencies
   import flash.events.Event;
   import flash.events.MouseEvent;
   import flash.display.MovieClip;
   import flash.text.TextField;

   // Declare our class
   public class Teaser extends MovieClip
   {
      // Constructor function.
      public function Teaser()
      {
         // Default to not selected.
         selected = false;

         addEventListener( Event.ADDED_TO_STAGE, onAdded );
         addEventListener( MouseEvent.MOUSE_OVER, setHover );
         addEventListener( MouseEvent.MOUSE_OUT, setNormal );
         addEventListener( MouseEvent.MOUSE_DOWN, setNormal );
      }

      // Called when this object has been added to the stage.
      private function onAdded( event:Event )
      {
         // Declare this teaser as a button.
         buttonMode = true;
         mouseChildren = false;

         // Go to the normal state.
         back.gotoAndStop("normal");
      }

      // Used to select a teaser.
      public function setSelected( _selected:Boolean )
      {
         selected = _selected;
         back.gotoAndStop( selected ? "selected" : "normal" );
      }

      // Called when the teaser is hovered over.
      private function setHover( e:MouseEvent )
      {
         if( !selected ) {
            back.gotoAndStop("hover");
         }
      }

      // Called when the mouse moves out of the teaser.
      private function setNormal( e:MouseEvent )
      {
         if( !selected ) {
            back.gotoAndStop("normal");
         }
      }
      public var back:MovieClip;
      public var title:TextField;
      public var selected:Boolean;
   }
}
```

The last and final change that we will need to make is to incorporate the node information from Drupal into our `Teaser` class. However, we run into a slight problem since the views service does not give us the full node information for each teaser, but rather just the node ID for each node within the list. Because of this, each teaser will need to have its very own `onNodeLoad` function that loads its own node information and then stores it within its own data structure. In order to fulfill this requirement, each node will need to have its own `Responder` to link to the `onNodeLoad` function for each instance of a Teaser that is created. Once the node has been loaded, we can then trigger a simple event that indicates to our `main.as` file that the node information for that teaser has been loaded. These changes will look as follows:

```
package
{
   // Import all dependencies
   import flash.events.Event;
   import flash.events.MouseEvent;
   import flash.display.MovieClip;
   import flash.text.TextField;
   import flash.net.Responder;
   // Declare our class
   public class Teaser extends MovieClip
   {
      // Constructor function.
      public function Teaser()
      {
         // Default to not selected.
         selected = false;
         // Setup our onNodeLoad responder.
         responder = new Responder( onNodeLoad, onError);
         addEventListener( Event.ADDED_TO_STAGE, onAdded );
         addEventListener( MouseEvent.MOUSE_OVER, setHover );
         addEventListener( MouseEvent.MOUSE_OUT, setNormal );
         addEventListener( MouseEvent.MOUSE_DOWN, setNormal );
      }
      // Called when this object has been added to the stage.
```

```
private function onAdded( event:Event )
{
   // Declare this teaser as a button.
   buttonMode = true;
   mouseChildren = false;

   // Go to the normal state.
   back.gotoAndStop("normal");
}

// Called when an error occurs.
function onError( error:Object )
{
   trace("An error has occurred!");
}

// Called when the node has been loaded.
public function onNodeLoad( _node:Object )
{
   node = _node;

   // Set the title.
   if( title ) {
      title.wordWrap = true;
      title.text = node.title;
   }

   dispatchEvent( new Event(Event.COMPLETE) );
}

// Used to select a teaser.
public function setSelected( _selected:Boolean )
{
   selected = _selected;
   back.gotoAndStop( selected ? "selected" : "normal" );
}

// Called when the teaser is hovered over.
private function setHover( e:MouseEvent )
{
   if( !selected ) {
      back.gotoAndStop("hover");
   }
}

// Called when the mouse moves out of the teaser.
private function setNormal( e:MouseEvent )
{
   if( !selected ) {
      back.gotoAndStop("normal");
   }
```

```
        }
        public var back:MovieClip;
        public var title:TextField;
        public var selected:Boolean;
        public var responder:Responder;
        public var node:Object;
    }
}
```

Now that we have a complete Teaser class, our next step is to build a mechanism to add these teasers to our application in List form.

Building a ListView class

Although there are many ways to present data in the form of lists within Flash, I have learned the hard way that making them work the way you want is something of a challenging task. Not to mention that the standard **List** component within Flash separates out all of the UI from the data, making the process of simply adding custom movie clips to that list extremely complicated. Because of this, we will do what any other programmer does when they can't find exactly what they are looking for…. build our own!

We will start this section by first creating a new ActionScript file called `ListView.as`, within our `chapter9` directory, and then start it out like any other class by creating a simple stub class that will simply contain the movie clips that will be contained within a **ListView** movie clip. These movie clips will be the **list** (which will act as the parent movie clip to all of our teasers), and the **listMask** (which will be used to only show a certain window of our list of teasers).

```
package
{
    // Import all dependencies
    import flash.events.Event;
    import flash.display.Sprite;
    // Declare our class
    public class ListView extends Sprite
    {
        // Constructor function.
        // Called when someone creates a new ListView
        public function ListView()
        {
            // Set up our events..
            addEventListener( Event.ADDED_TO_STAGE, onAdded );
        }
```

```
      // Called when this object has been added to the stage.
      private function onAdded( event:Event )
      {
      }
      // Delcare our child movie clips.
      public var list:Sprite;
      public var listMask:Sprite;
   }
}
```

We now need a mechanism to add our teasers to the **ListView** using an `addItem` function. Within this function, we will keep track of the previously added teaser position, and then position the new teaser directly below the previously added teaser. This functionality will look like the following:

```
package
{
   // Import all dependencies
   import flash.events.Event;
   import flash.display.Sprite;
   import flash.geom.Rectangle;
   // Declare our class
   public class ListView extends Sprite
   {
      // Constructor function.
      // Called when someone creates a new ListView
      public function ListView()
      {
         // Instanciate our variables.
         lastRect = new Rectangle(0,0,0,0);
         // Set up our events..
         addEventListener( Event.ADDED_TO_STAGE, onAdded );
      }
      // Called when this object has been added to the stage.
      private function onAdded( event:Event )
      {
         // Remove the default teaser.
         list.removeChildAt( 0 );
      }
      // Used to add an item to our list view.
      public function addItem( item:* )
      {
         // Set the items position in the list.
         item.y = lastRect.y = lastRect.y + lastRect.height;
         item.x = lastRect.x;
         lastRect.width = item.width;
         lastRect.height = item.height;

         // Add the item to the list.
         list.addChild( item );
      }
      // Delcare our child movie clips.
      public var list:Sprite;
      public var listMask:Sprite;

      // The previously added teasers rectangle region.
      private var lastRect:Rectangle;
   }
}
```

And now, our final task is to add the scroll functionality to this list view. To simplify this process, we will make our **ListView** automatically scroll when the user hovers over the scroll region. Instead of explaining all the steps to incorporate this change, we can follow each change by observing the comments made for each change to this file.

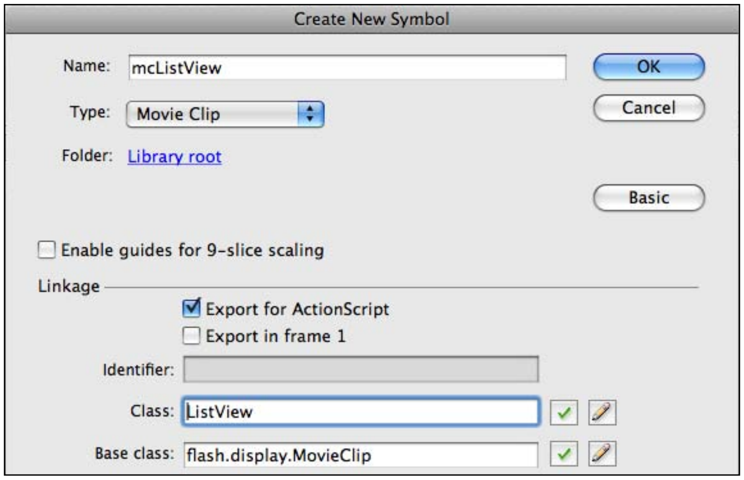
```
package
{
   // Import all dependencies
   import flash.events.Event;
   import flash.events.MouseEvent;
   import flash.display.Sprite;
   import flash.geom.Rectangle;

   // Declare our class
   public class ListView extends Sprite
   {
      // The scroll speed of our list view.
      private static const speed:Number = 10;

      // The hysteresis of our scroll bar.
      private static const hyst:Number = 15;

      // Constructor function.
      // Called when someone creates a new ListView
      public function ListView()
      {
         // Instanciate our variables.
         lastRect = new Rectangle(0,0,0,0);

         // Set up our events..
         addEventListener( Event.ADDED_TO_STAGE, onAdded );
         addEventListener( MouseEvent.MOUSE_OVER, onOver );
         addEventListener( MouseEvent.MOUSE_OUT, onOut );
      }
      // Called when this object has been added to the stage.
      private function onAdded( event:Event )
      {
         // Get the scroll mid point.
         scrollMid = listMask.height / 2;
         // Remove the default teaser.
         list.removeChildAt( 0 );
      }
      // Called when the mouse moves over our list.
      private function onOver( e:MouseEvent )
      {
         // Trigger on every frame event.  30 frames per sec.
         addEventListener( Event.ENTER_FRAME, scrollEvent );
      }
      // Called when the mouse moves out of our list.
      private function onOut( e:MouseEvent )
      {
         // Remove the trigger event.
         removeEventListener( Event.ENTER_FRAME, scrollEvent );
      }
      // Scroll's our list region.
      private function scrollEvent( e:Event ) : void
      {
         // Get our relative mouse position to the mid point.
         var mousePos:Number = listMask.mouseY - scrollMid;
         // See if our list height is greater than the mask.
         var shouldScroll:Boolean = (list.height > listMask.height);
         // See if we are not within our hysteresis region.
         shouldScroll = shouldScroll && (Math.abs(mousePos) > hyst);
         // If we should scroll.
         if( shouldScroll ) {
            // Find the delta.
            var delta:int = speed * (mousePos / scrollMid);
            // Set our list position.
            list.y = list.y - delta;
            // Make sure the list does not scroll down to far.
            list.y = (list.y > 0) ? 0 : list.y;
            // Make sure the list does not scroll up to far.
            var bot:Number = -(list.height - listMask.height);
            list.y = (list.y < bot) ? bot: list.y;
         }
      }
      // Used to add an item to our list view.
```

```
         public function addItem( item:* )
         {
            // Set the items position in the list.
            item.y = lastRect.y = lastRect.y + lastRect.height;
            item.x = lastRect.x;
            lastRect.width = item.width;
            lastRect.height = item.height;

            // Add the item to the list.
            list.addChild( item );
         }

         // Delcare our child movie clips.
         public var list:Sprite;
         public var listMask:Sprite;

         // The previously added teasers rectangle region.
         private var lastRect:Rectangle;

         // The midpoint of the list mask.
         private var scrollMid:Number;
      }
   }
```

Now that we have our **ListView** control ready for action, we can make some changes to our `chapter9.fla` file so that it incorporates out new **ListView** control.

Adding our ListView to Flash

With our `chapter9.fla` project file open, we can start our task of creating a list view by clicking on the menu item **Insert | New Symbol**, where we will then provide the following information:



Create New Symbol

Name: mcListView

Type: Movie Clip

Folder: Library root

□ Enable guides for 9-slice scaling

Linkage

☑ Export for ActionScript

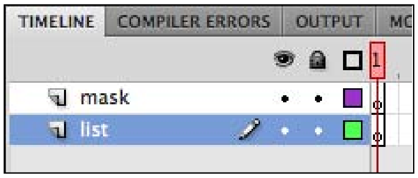
□ Export in frame 1

Identifier:

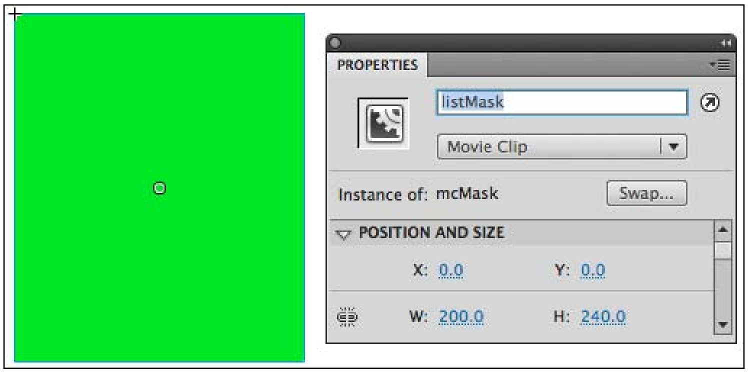
Class: ListView

Base class: flash.display.MovieClip

After we click on **OK**, we should be within our new movie clip, where we will first set up two different layers, one for the list and one for the list mask as follows:



Within the **mask** layer, we will simply draw a rectangle region that will act as the visible window for our teaser list. Since this will be a mask region, we can use a crazy green fill color with no border, and then set the rectangle to have the dimensions of 200 pixels wide (same as our teasers) and 240 pixels high (same as our video). When we are finished drawing our mask rectangle, we can then turn it into a movie clip called **mcMask** with an instance name of **listMask** (as referenced within our **ListView** class).



listMask

Movie Clip

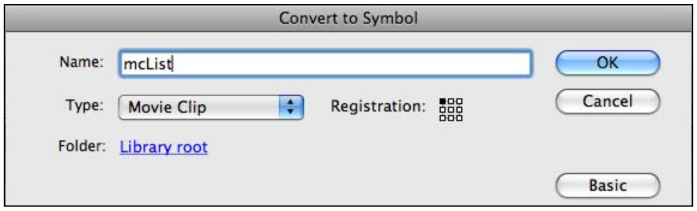
Instance of: mcMask

X: 0.0 Y: 0.0

W: 200.0 H: 240.0

We can now lock the **mask** layer, and then select the **list** layer where we will first drag and drop our **mcTeaser** movie clip from the **Library** onto our stage, making sure we give it an x and y coordinate of (0,0).

Once the teaser has been added to our stage, we will simply click on the **mcTeaser** movie clip and then select the menu item **Modify | Convert to Symbol**, where we will give our new symbol a name of **mcList**.



Name: mcList

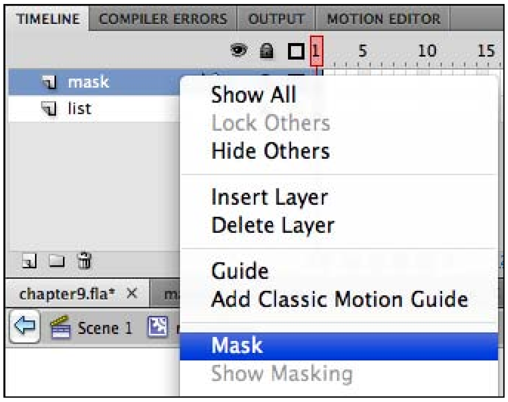
Type: Movie Clip

Registration:

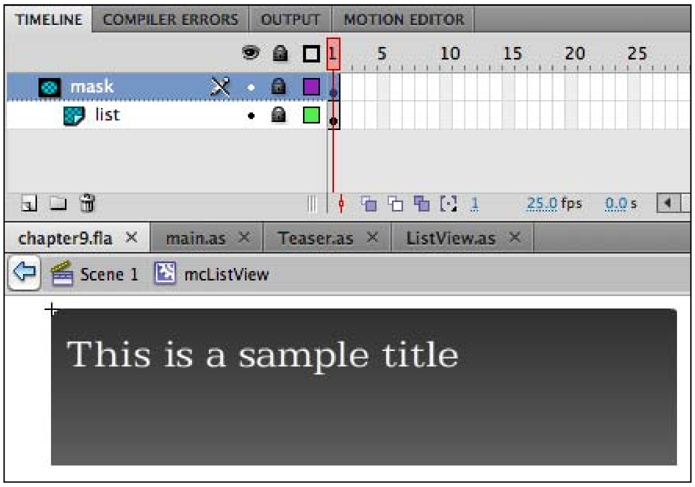
Folder: Library root

Once our new movie clip has been created, we will give it an instance name of **list**.

We can now set up our **mask** layer, so that it behaves as a mask by first locking both the **list** and **mask** layers. Once they are locked, we can then right-click on the **mask** layer and select **Mask** from the drop-down menu.



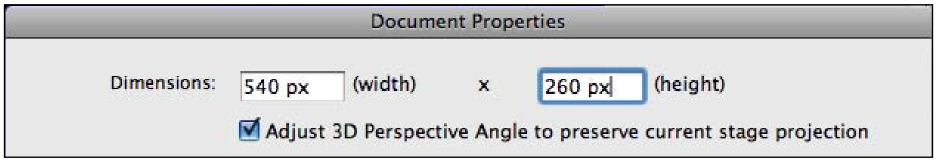
We should then see something similar to the following:



We are now ready to move on and incorporate this **ListView** into our main player, so that it shows a listing of all teasers within our Drupal view.

Adding the ListView to our Media Player

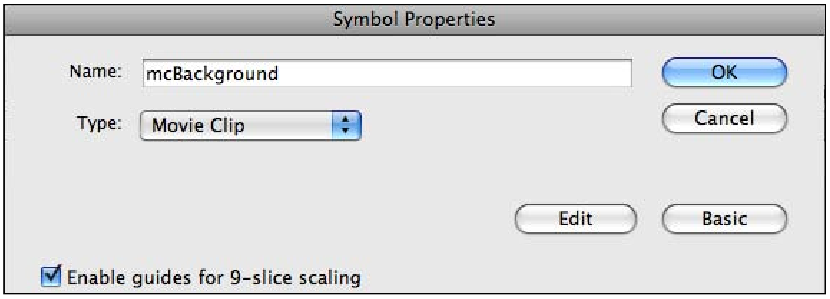
Now that we have a ListView control, we can add this list view to our media player by first adding the **mcListView** to the main stage of our `chapter9.fla` project. With this project open, we will first want to resize our stage so that it can hold our new playlist by clicking on the stage, and then clicking on the **Edit** button next to the stage dimensions in the **Properties** section. We can then resize our stage to a new size of 540 pixels wide and 260 pixels high. This will allow us to have a 10-pixel border with a 320x240 video with a 200x240 playlist right next to it.


Document Properties

Dimensions: 540 px (width) x 260 px (height)

☑ Adjust 3D Perspective Angle to preserve current stage projection

Now, with the stage set to a larger size, our next task is to resize our background so that it matches the width and height of our stage. We can do this in the same way that we did in Chapter 3, where we first created a new movie clip for our background, and then used the 9-slice scaling for that movie clip to resize our background without distortion.

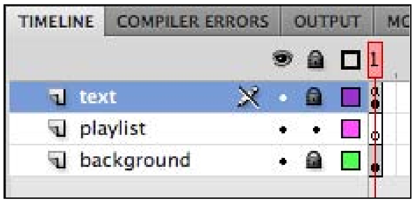

Symbol Properties

Name: mcBackground

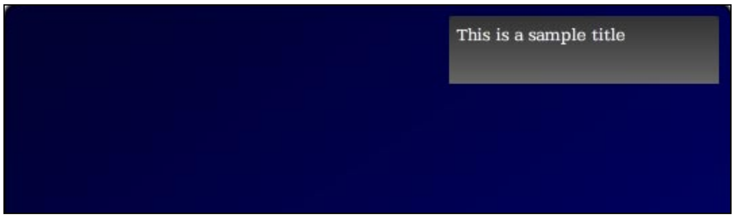
Type: Movie Clip

☑ Enable guides for 9-slice scaling

Once we have done this, we can create a new layer to hold our **ListView** above our background layer.



We can now drag and drop our **mcListView** from the Library and onto the stage. We will then set its X and Y position on the stage to 331px and 10px respectively.

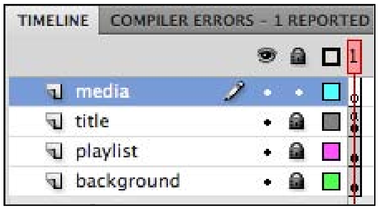


This is a sample title

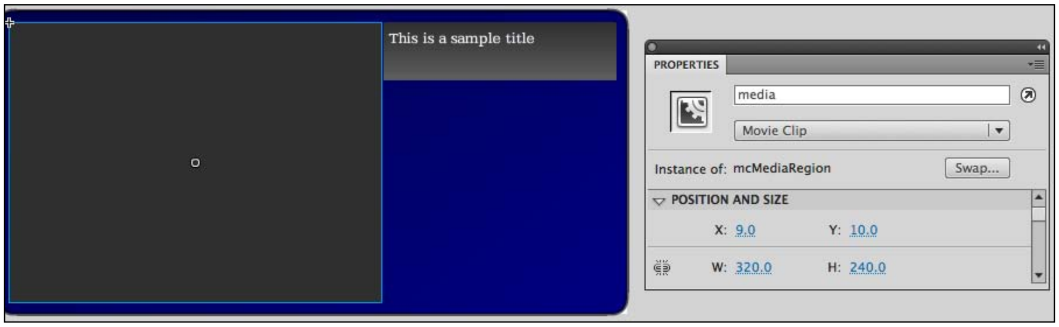
Now that we have our views playlist set up on our stage, our next task is to define a media region that will hold our video.

Creating a Media Region

In Chapter 6, we created a video player that was designed to take up the width and height of the entire stage. Since we now have a video playlist and a background region, we will need to make a change to our project so that we can define the width and height of our video to take up a designated area of our player. To do this, we will first create a new layer in our project that will hold our media region as shown:



Within this layer, we will create a new dark grey rectangle that is 240x320, and then create a new movie clip from that region called **mcMediaRegion**. Once we have this new media region created, we will give it an instance name of **media** as shown:



This is a sample title

PROPERTIES

media

Movie Clip

Instance of: mcMediaRegion

POSITION AND SIZE

X: 9.0 Y: 10.0

W: 320.0 H: 240.0

Our next task is to modify our `main.as` file so that we add the media to this media movie clip rather than the stage as seen below.

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
   // Declare our base URL.
   var fileURL:String = baseURL;
   // Add our file's relative path.
   fileURL += "/";
   // Declare a generic media player.
   var player = null;
   // If this node has audio.
   if( node.audio ) {
      // Declare our player as an AudioPlayer.
      fileURL += node.audio.file.filepath;
      player = new AudioPlayer();
   }
   else if( node.field_video ) {
      // Declare our player as a VideoPlayer.
      fileURL += node.field_video[0]["filepath"];
      player = new VideoPlayer();
   }
   // Add the player to the media region.
   media.addChild( player );
   // Add a control bar.
   addControlBar();
   // Play our audio file
   player.load( fileURL );
}
```

Finally, our last change will be to the `VideoPlayer.as` file, where we will set the width and height of our player to the width and height of the parent movie clip rather than the entire stage.

```
// Called when the video player has been added to the stage.
private function onAdded( event:Event )
{
   // Create our video object the size of our parent.
   video = new Video( parent.width, parent.height );
   // Attach our net stream to the video object.
   video.attachNetStream( stream );
   // Add the video to the VideoPlayer.
   addChildAt( video, 0 );
}
```

Now that we have our media region ready, our next task will be to edit the `main.as` file so that our list region will populate a new teaser for every node within our view.

Populating the list view

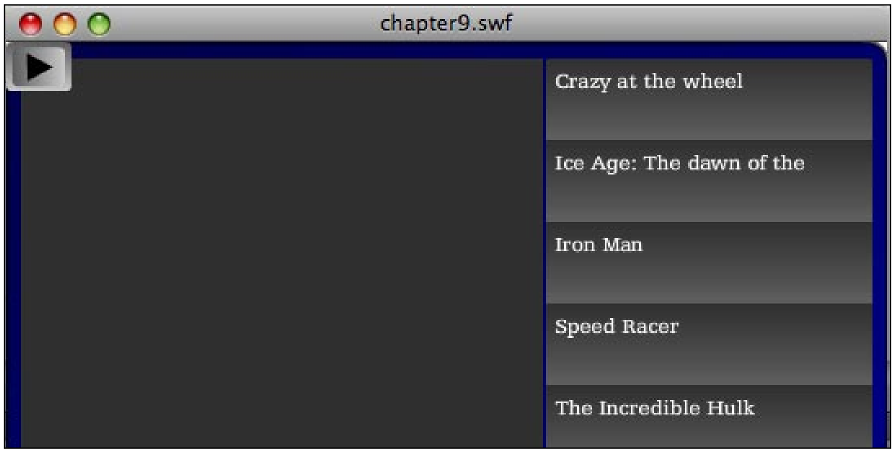
With the `main.as` file open, we can now pick up where we left off in the previous section by editing the `onViewLoad` function so that it creates a new Teaser for each new node and then adds those teasers to the **ListView**.

```
// Called when a view gets loaded.
function onViewLoad( _view:Object )
{
   // Iterate through all of the nodes.
   for each( var node:Object in _view )
   {
      // Declare our new teaser.
      var teaser:Teaser = new Teaser();
      // Add our teaser to the list.
      view.addItem( teaser );
   }
}
```

If we were to run our video player at this point, we should be happily surprised to see that a new teaser is added to our **ListView** for every new teaser that is given in our view. We can now load the node data, within each teaser, by calling `node.get` on each teaser that is added to our list.

```
// Called when a view gets loaded.
function onViewLoad( _view:Object )
{
   // Iterate through all of the nodes.
   for each( var node:Object in _view )
   {
      // Declare our new teaser.
      var teaser:Teaser = new Teaser();
      // Add our teaser to the list.
      view.addItem( teaser );
      // Load the node information into the teaser.
      drupal.call( "node.get", teaser.responder, sessionId,
      node.nid );
   }
}
```

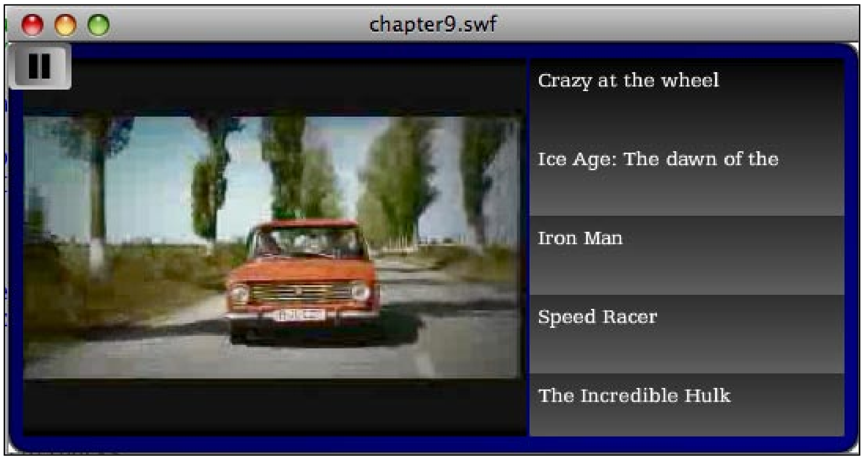
Running the application again, we should see that all the teaser information is populated within our playlist.



Moving on, we will now need to hook up the Click and Load events from our Teaser to interact with our main media region. To do this, we will simply need to register for two events from our Teaser, one when it is clicked with the mouse, and the other when the teaser has finished loading its node information. We will also need to keep track of which teaser is selected so that we can de-select it when a new one is selected. Once we make all of our changes to the `main.as` file, it should look like the following.

```
// Declare our variables
var baseURL:String = "http://localhost/drupalbook";
var gateway:String = baseURL + "/services/amfphp"
var sessionId:String = "";
var nodeId:Number = root.loaderInfo.parameters.node;
var viewName:String = "videos";
var id:String = root.loaderInfo.parameters.id;
var connect:String = root.loaderInfo.parameters.connect;
var selectedTeaser:Teaser = null;
...
...
// Called when a view gets loaded.
function onViewLoad( _view:Object )
{
   // Iterate through all of the nodes.
   for each( var node:Object in _view )
   {
      // Declare our new teaser.
      var teaser:Teaser = new Teaser();
      // Add an event handler when this teaser is clicked.
      teaser.addEventListener( Event.COMPLETE, onTeaserLoad );
      teaser.addEventListener( MouseEvent.MOUSE_UP, onTeaserClick );
      // Add our teaser to the list.
      view.addItem( teaser );
      // Load the node information into the teaser.
      drupal.call( "node.get", teaser.responder, sessionId,
      node.nid );
   }
}
...
...
// Called when a teaser loads
function onTeaserLoad( e:Event )
{
   if( !selectedTeaser ) {
      loadTeaser( (e.target as Teaser) );
   }
}
// Called when a teaser gets selected.
function onTeaserClick( e:MouseEvent )
{
   // Load the selected teaser.
   loadTeaser( e.target as Teaser );
}
// Function to load any teaser.
function loadTeaser( teaser:Teaser )
{
   // Deselect the selected teaser.
   if( selectedTeaser ) {
      selectedTeaser.setSelected( false );
   }
   // Set the new teaser and select it.
   selectedTeaser = teaser;
   selectedTeaser.setSelected( true );
   // Load the node.
   onNodeLoad( selectedTeaser.node );
}
// Called when an error occurs connecting to Drupal.
function onError( error:Object )
{
   for each (var item in error) {
      trace(item);
   }
}
```

When we run this application, we should have a fully functional media player with a playlist driven by the power of Drupal Views!



Summary

In this chapter we covered a lot of ground by building a media player whose playlist is driven from the power of the Drupal Views module. After walking through this exercise, it is probably easily understood how powerful this feature is when building dynamic lists of content in our Flash application. Since we can control and govern all of the filtering for our content using the Drupal administrator backend, we can essentially customize our Flash applications without having to recompile our SWF files. In my opinion, this is the most powerful advantage of using a Content Management System over the traditional static XML-driven content.

In the next chapter, we will take our interaction with Drupal one step further by learning how to add content to a Drupal site from a Flash application.

10
User Management

In the previous chapters we learned the essentials of extracting Drupal data to be used within a remote Flash application. Since most of that data was delivered as read-only, not much effort was needed to help secure the integrity of the information that resided on the hosting server. Our goal for this chapter, along with the following chapters, is to explore how that data can be manipulated and controlled from a remote location. We will learn how a remote Flash application can use the data from a remote location. We will also learn how to manipulate and change that data. Obviously, this raises many security questions on how we can allow this type of control while at the same time keep our data safe from malicious software. And the answer for those questions begins with the integration of Drupal's user management system, which is paramount in protecting and managing all the data within its database. By utilizing Drupal's built-in user management system, any remote application can literally log into the Drupal web site, which in turn validates that user to perform any data manipulation task.

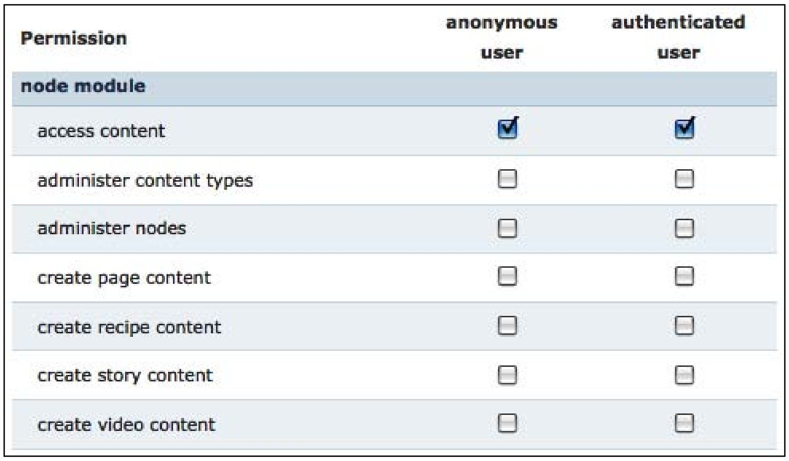
In this chapter we will tackle the first part of the data manipulation puzzle by building a user login block. This user login block can be used within our Flash applications to log into Drupal from a remote location. We will accomplish this goal by walking through the following key topics:

- Drupal user management
- The User Service module
- Building a Flash user login block
- User handling within Flash
- Logging into Drupal from Flash

Drupal user management

Before we dive into building a login system in Flash, we must first understand how Drupal protects our data using its built-in user management system. This user management system is extremely vital to the security of our Drupal system as it constantly checks if each user (even the anonymous ones) has access to do his or her intended task. And at the core of this data security system, is a permission-based system governed by the user roles.

All users visiting our Drupal web site are assigned a specific role, where they can perform specific tasks, if their role allows such operations. Anything from viewing content to changing module settings is tightly controlled using the Drupal user role system. We can see this firsthand by logging into our administrator Drupal account and then navigating to the **Permissions** section within our Drupal administrator backend. Here, we should see some permission settings for each module that might resemble the following:



Permission	anonymous user	authenticated user
node module		
access content	☑	☑
administer content types	□	□
administer nodes	□	□
create page content	□	□
create recipe content	□	□
create story content	□	□
create video content	□	□

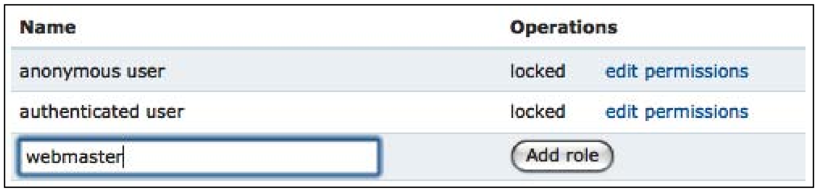
What this image is showing is how the Drupal administrator can easily control who is allowed to do certain things by checking the checkbox for that permission in the column for that specific user role. If a box is not checked, then the default action is to deny that user role access to that function.

By default, there are only two different user roles; **anonymous** pertains to any user who is not logged into the Drupal system, while **authenticated** pertains to any user who has registered and is logged into the Drupal web site. Although this may not seem like much control for specific user permissions, the true power comes from the administrator having the ability to add as many user roles as they like to control the access levels to an even more refined level.

Adding new user roles

Adding a new user role is a very simple process. We will start by deciding what user role we would like to create, which should illustrate how this process works. In the next chapter, we will create a Flash widget that will be able to create new web pages in our Drupal system. However, the current user roles do not allow the creation of user pages since the **create page content** permission is unchecked for both user roles. Instead of enabling this permission for all authenticated users, we would like to have the ability to refine this privilege to only certain users who we will call **webmasters**. To start this off, we will need to create a new user role called **webmaster** by navigating to **Administer | Roles**.

Once we are in the **User Roles** section of the Drupal administrator, we can easily add a new role by typing **webmaster** in the text field underneath the last role, and then clicking on the **Add role** button.



Name	Operations	
anonymous user	locked	edit permissions
authenticated user	locked	edit permissions
webmaster	Add role	

Now that we have successfully added a new **webmaster** role to our Drupal system, the next step in this process is to add permissions to this user role.

Adding permissions to a user role

To edit the permissions for a user role, we can either navigate to the **Permissions** section in the Administrator section, or we can simply click on the **edit permissions** link next to our new user role. Either way, we should see the **User Permissions** section, where we can add new permissions to our **webmaster** role.

To start, we will check all the checkboxes that are checked for the **authenticated** role, but within our **webmaster** column. This will give us all the same permissions as any other authenticated user, but we will add to the **webmaster** permissions by checking other permissions that we want to enable for this role. In addition to the **authenticated** role permissions, we will also need to make sure the following permissions are checked for the **webmaster** role:

- **create page content**
- **create story content**
- **delete own page content**

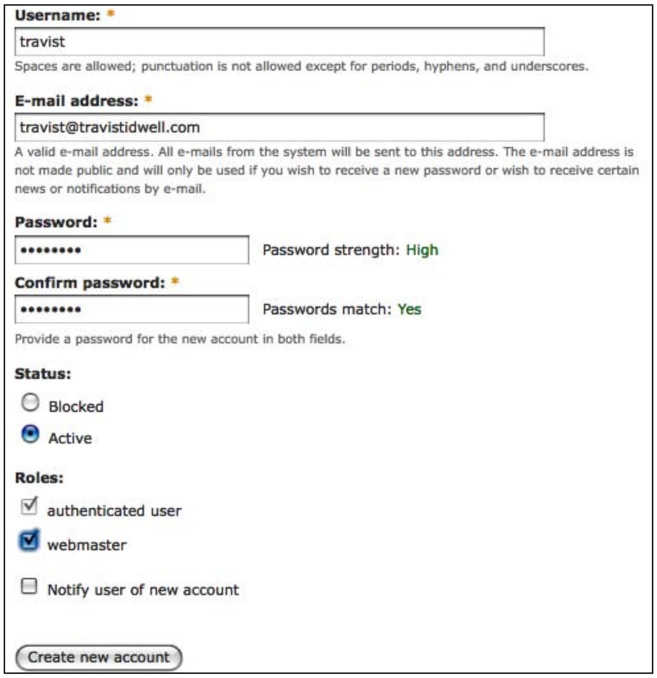
- **delete own story content**
- **edit own page content**
- **edit own story content**

What we have just done is allow any user with the **webmaster** role to create new page and book content onto our Drupal server. However, we will not really be able to test this until we create a new user who has this new **webmaster** role.

Creating new users and assigning them roles

Our final task with setting up the user management for our Flash application will be to add a new **webmaster** user who we will use to add content to Drupal from within Flash. To do this, we will first need to navigate to the users' section within the Drupal administrator by going to **Administer | Users** and clicking on the button that says **Add User**.

Here, we will enter a new **Username** and **E-mail address**. Make sure to check the **webmaster** user role. Once we have our new user filled out, we can create the user by clicking on the **Create new account** button.



Username: *
travist
Spaces are allowed; punctuation is not allowed except for periods, hyphens, and underscores.

E-mail address: *
travist@travistidwell.com
A valid e-mail address. All e-mails from the system will be sent to this address. The e-mail address is not made public and will only be used if you wish to receive a new password or wish to receive certain news or notifications by e-mail.

Password: *
•••••••• Password strength: High

Confirm password: *
•••••••• Passwords match: Yes
Provide a password for the new account in both fields.

Status:
○ Blocked
● Active

Roles:
☑ authenticated user
☑ webmaster

□ Notify user of new account

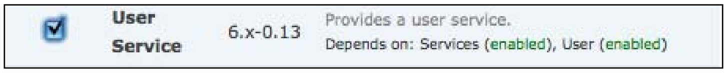
And congratulations, we now have a new user ready to add data to our Drupal web site! Our next task will be to install and configure the **User Service** module.

The User Service module

The User Service module is a Services plug-in that acts as the glue between Drupal's user management system and remote applications requiring the ability to log into our web site. Giving remote applications the power to log into our Drupal web site opens up a whole slew of possibilities, where user-specific data can be presented in a secure manner to those remote applications. Not only that, we can design our applications so that content can be added, edited, and removed from our Drupal site from a remote Flash application. Of course, this depends on whether the user logged in to our Flash application has the privileges to do such operations. We will explore this great feature by first installing and configuring the User Service.

Installing the User Service

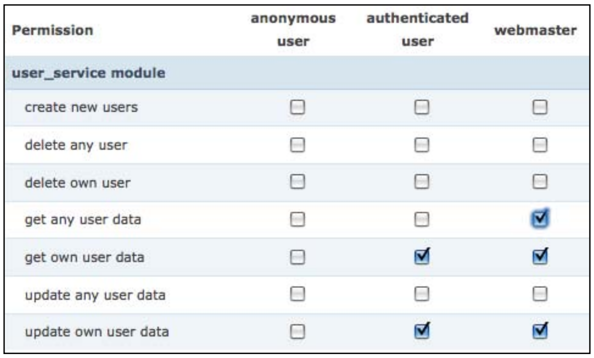
Since we have already downloaded the Services module, we will not have to do much in this section simply because the User Service module comes pre-packaged with the Services module. Because of this, we can install the user service much like any other module in our system; by navigating to the Drupal modules section and then clicking on the checkbox next to the **User Service** module within the Services section. Once we have done this, we can install this module by clicking the **Save Configuration** button at the bottom of the page.



☑	User Service	6.x-0.13	Provides a user service. Depends on: Services (enabled), User (enabled)

Configuring permissions

Now that we have enabled the user service module, our next task is to navigate to the **Permissions** section (**Administer | Permissions**), where we will select the permissions that will be required to utilize this module from a remote source. By scrolling down to the **user_service module** section, we should set up our permissions to be as follows:

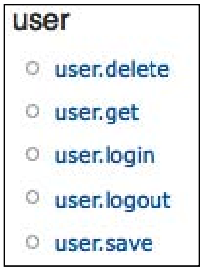


Permission	anonymous user	authenticated user	webmaster
user_service module			
create new users	☐	☐	☐
delete any user	☐	☐	☐
delete own user	☐	☐	☐
get any user data	☐	☐	☑
get own user data	☐	☑	☑
update any user data	☐	☐	☐
update own user data	☐	☑	☑

After we check the following permissions, we must now make sure to commit this change by clicking on the button at the bottom of the page that says **Save permissions**.

Configuring the User Service module

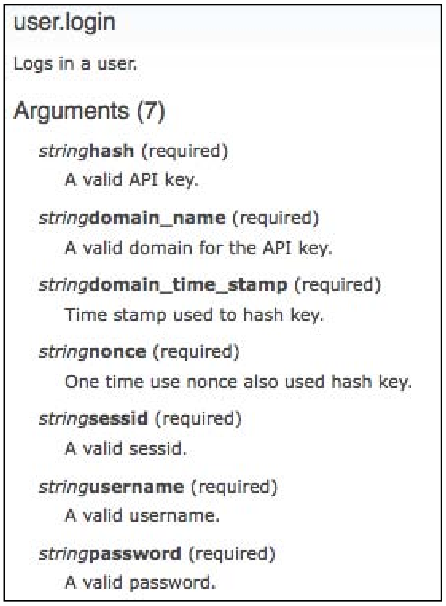
Our next task is to take a journey to the **Services Administrator** (by navigating to **Administer | Services**), where we should see our new service functions available to our remote applications.



user

- user.delete
- user.get
- user.login
- user.logout
- user.save

Although each of these services can be very useful, we will only be utilizing the **login** and **logout** services in this chapter to illustrate how this module works. We can see how the interface looks for each of these functions by clicking on the links defined by each service call (**user.delete**, **user.get**, and so on). Let's do this by clicking on the **user.login** service method, where we should see the following:



user.login

Logs in a user.

Arguments (7)

*string***hash** (required)
A valid API key.

*string***domain_name** (required)
A valid domain for the API key.

*string***domain_time_stamp** (required)
Time stamp used to hash key.

*string***nonce** (required)
One time use nonce also used hash key.

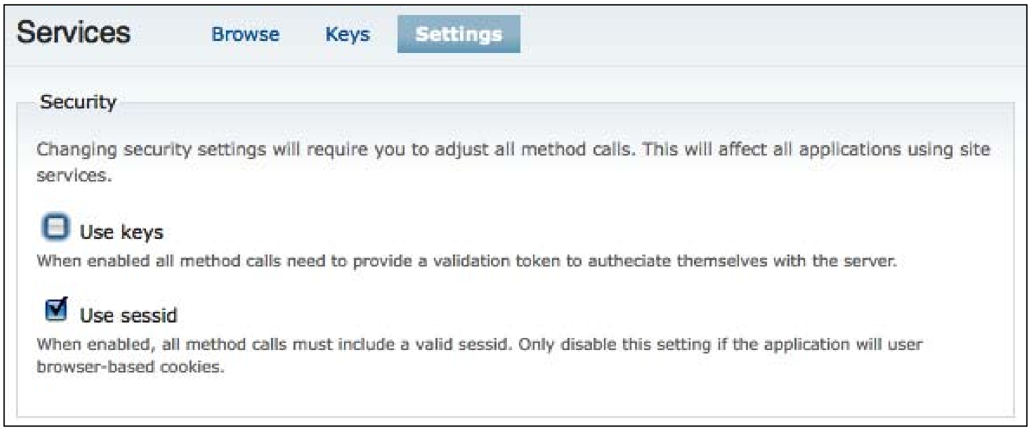
*string***sessid** (required)
A valid sessid.

*string***username** (required)
A valid username.

*string***password** (required)
A valid password.

At first glance, anyone looking at all of these arguments would most likely be very intimidated. The majority of these arguments, except for the last two, are used for security reasons and much explanation is required on what they are and how they are used. We will cover all of this information in the next chapter. For this chapter, we will do something that I would normally not recommend: disable the API key so that we can get a working login system without all of these security arguments. Then, once we walk through the next chapter, we will re-enable the API key to help minimize any security issues that may threaten our web site.

With that said, we can now disable the API keys by clicking on the **Settings** tab within the Services Administrator and then uncheck the checkbox where it says **Use Keys**.



Security

Changing security settings will require you to adjust all method calls. This will affect all applications using site services.

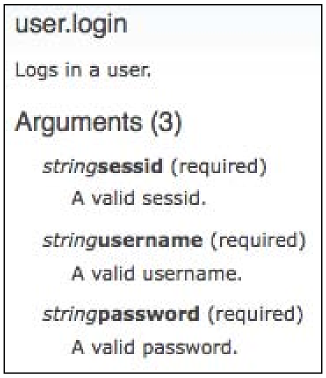
□ Use keys

When enabled all method calls need to provide a validation token to autheciate themselves with the server.

☑ Use sessid

When enabled, all method calls must include a valid sessid. Only disable this setting if the application will user browser-based cookies.

Now, when we navigate back to our **user.login** function declaration, we will be happily surprised to see that all that is required to log into our Drupal system from a remote application is the session ID (which we know), along with the **username** and **password** of that user.



user.login

Logs in a user.

Arguments (3)

*string***sessid** (required)
A valid sessid.

*string***username** (required)
A valid username.

*string***password** (required)
A valid password.

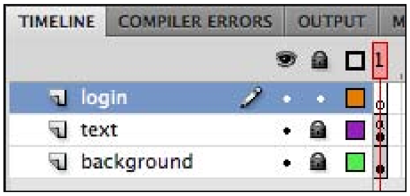
> Disabling the API keys is ONLY done to simplify the flow of this chapter and is not recommended as a permanent solution. It is highly recommended to complete this chapter and read the next chapter where security is discussed and where the API keys are re-enabled.

Now that we have our User Service in place, we can shift our focus back to Flash, where we will build a user login system.

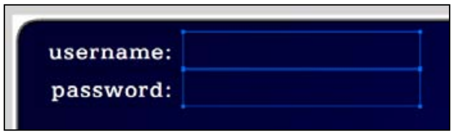
Building a Flash user login block

Our first task in this section will be to copy all of the Flash contents from the `chapter2` folder, and then, copy all of the contents of that folder to a new folder called `chapter10`. We will rename the `chapter2.fla` project file to `chapter10.fla`. The reason we are starting with Chapter 2 is simply because any of the additional functionality that was added to the chapters following the second will be white noise to our real goal at hand, which is to create a user login system.

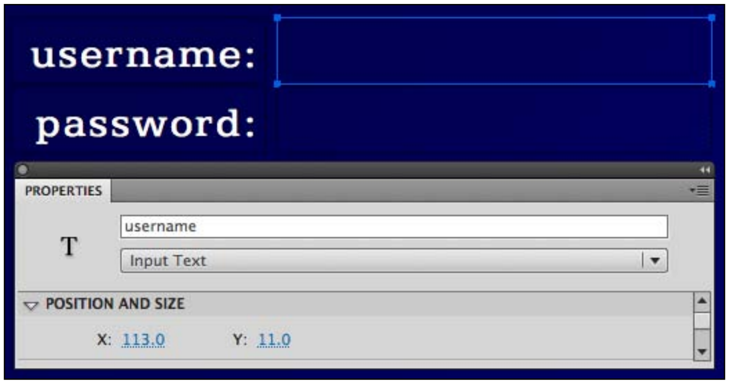
Once we have our new chapter directory, we will open up the `chapter10.fla` project file, where we will construct our user login. With this project open, we will start out by creating a new layer in our timeline that will be used for our user login block.



Once we have this layer selected, we can now construct our user login by first clicking on the T tool, and then placing two Input Text fields on the top lefthand corner of our stage along with two static text fields that will be used as the labels.



After we have our text fields in place, we will now want to give each of them an instance name so that they can be referenced within ActionScript. We will give these text fields an instance name of **username** and **password** respectively.

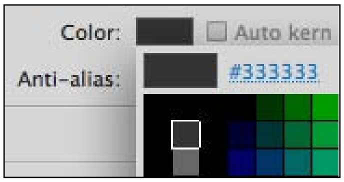


username

Input Text

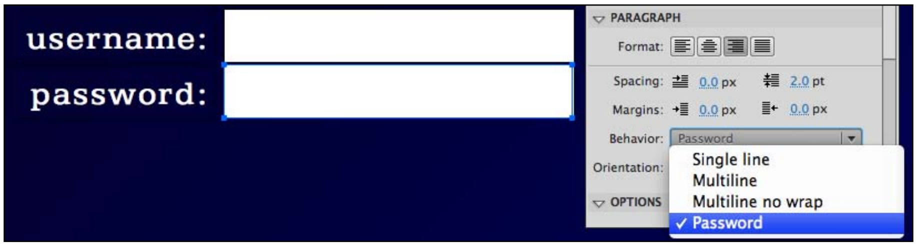
X: 113.0 Y: 11.0

Now that each of these fields has been created, we will need to make sure that the user knows where to type in their username and password. By default, the **Input Text** field does not set the background color as white, which would make it hard for the user to see where to start typing. We can fix this by giving our text fields a border by selecting each one of the **Input Text** fields and then clicking on the ▤ button within the **Properties** panel. We will also need to make sure that the text color within these text fields is a color that is not the same as the background color of our Input field. Otherwise, we will get a surprise when we try to enter text and it appears that nothing is happening, even though text is being entered but is just not visible. We can do this by clicking on each of the input fields and then, from **Color** within the **Properties** panel, selecting a dark color.



#333333

Finally, for the password field, we will need to make sure that all the characters are hidden when the user types in their password. We can do this by selecting the password text field and then selecting **password** from the **Behavior** drop-down box within the **Properties** panel.

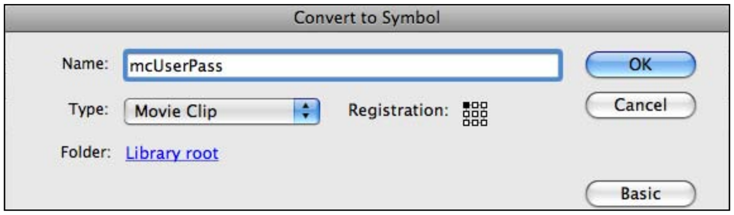


Spacing: 0.0 px 2.0 pt

Margins: 0.0 px 0.0 px

Behavior: Password

Now that we have these two fields in place, our next task will be to create a new movie clip that will act as a container for the username and password text fields along with their labels. To do this, we will need to select both the **username** and **password** input text fields along with their labels and then select **Modify | Convert to Symbol**, where we will create a new **Movie Clip** called **mcUserPass**.



Convert to Symbol

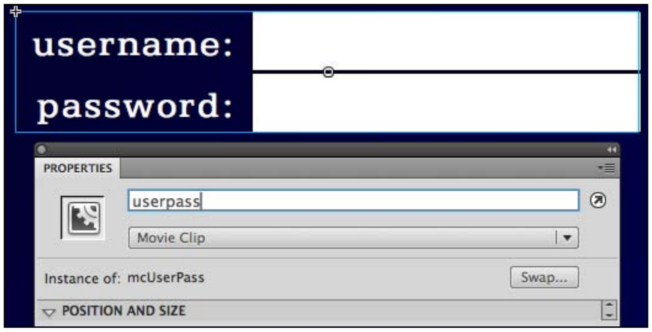
Name: mcUserPass

Type: Movie Clip

Registration:

Folder: Library root

Once this movie clip has been created, we will give it an instance name of **userpass** so that we can reference it within ActionScript.



userpass

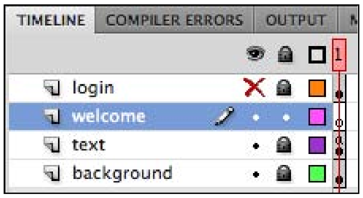
Movie Clip

Instance of: mcUserPass

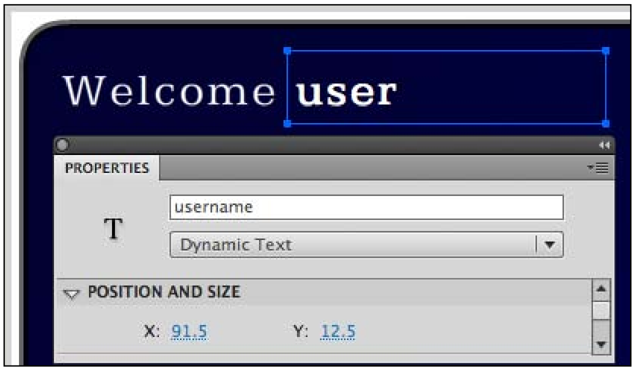
Now that we have a user login block, we still need to handle the situation when the user has already logged in.

Welcoming our logged-in users

For this section, we will want to greet our users who are logged in by simply showing a text field that says "Welcome" followed by the name of that user. We can start this by first creating a new layer below the **login** layer called **welcome**, and then hiding and locking the **login** layer so that it does not get in our way.



With the welcome layer selected, we can now add our text strings on the stage in the same spot as the **userpass** movie clip, if it were visible. This welcome message will simply consist of two different text fields, one **Static** and the other one **Dynamic**. The static text field will simply display "Welcome", while the dynamic text field will be used to dynamically populate our user's name. Because of this, we will need to make sure we give it an instance name of **username**.

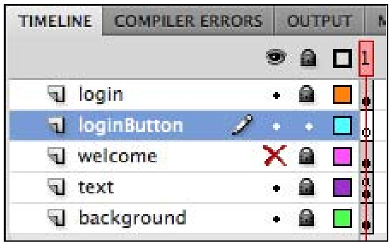


Once we are done setting up our welcome text fields, we can follow the steps given earlier in this section to create a new movie clip from the text fields called **mcWelcome**, with an instance name of **welcome**.

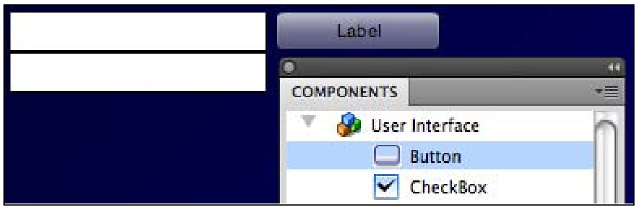
Now that we have our **welcome** layer, we will want to lock and hide the welcome layer, and then reshow the **login** layer so that we can accurately place our login button.

Creating a login button

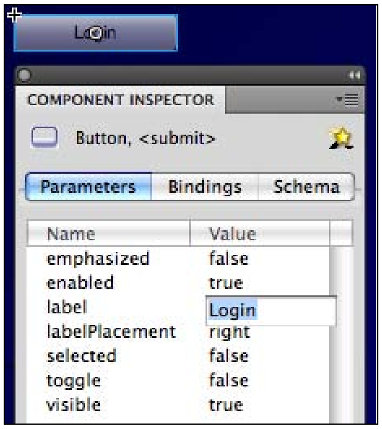
Our next step is to create a login button, so that the user has control over logging in or logging out of our Flash application. We can do this by first creating a new layer below our **login** layer called **loginButton**.



Then, within the **loginButton** layer, we can use the pre-defined **Button** component from the **Components** section of Flash to place our new login button.



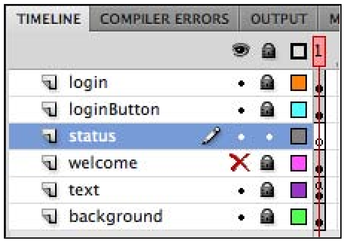
With our login button in place, our last task, within Flash, is to give our login button an instance name of **submit**, and then change the text within that button. To change the text within the button, simply click on the button and open up the **Component Inspector** by clicking on the **Window | COMPONENT INSPECTOR** menu, and then changing the text inside the **label** field.



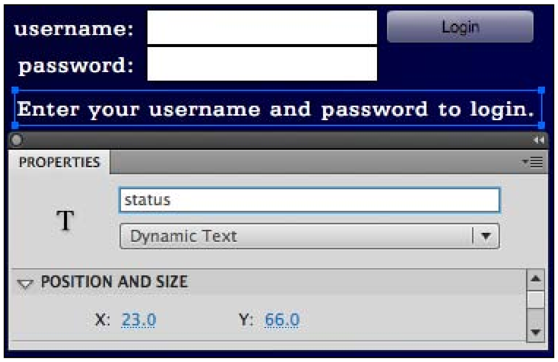
Name	Value
emphasized	false
enabled	true
label	Login
labelPlacement	right
selected	false
toggle	false
visible	true

Adding some status text

Now that we have a login button, our last element that we will need for our login block is a status text box that will tell the user if an error occurred. This can be done pretty easily by creating a new layer in their timeline called **status.**



Now that we have our own status layer, we will create a new dynamic text field within this layer, with an instance name of **status**.



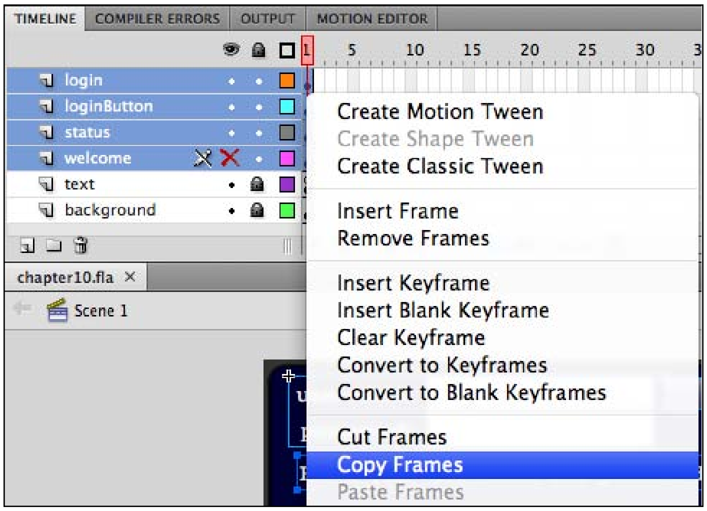
username:
password:
Enter your username and password to login.

PROPERTIES
status
Dynamic Text
POSITION AND SIZE
X: 23.0 Y: 66.0

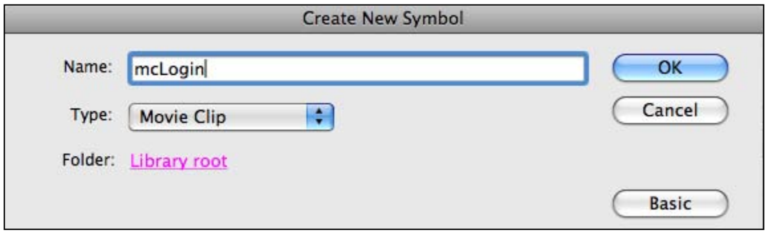
We should now have a set of Flash objects that resemble a user login block. To keep everything componentized, our next step will be to place all of the login components that we just created into their own movie clip called mcLogin.

Creating a mcLogin movie clip

Since our goal here is to retain all of the layers and objects that we just created, we will need to take a different approach when combining these elements into a single movie clip. We will start this process by first copying all of the layers that we just created for our login block by unlocking these layers, and then, using the *Shift* key, selecting all of them as a group. When they are all selected, we can then right-click and select **Copy Frames**.



With these frames copied, we can now create an empty movie clip called **mcLogin** by selecting **Insert | New Symbol** in the Flash menu.



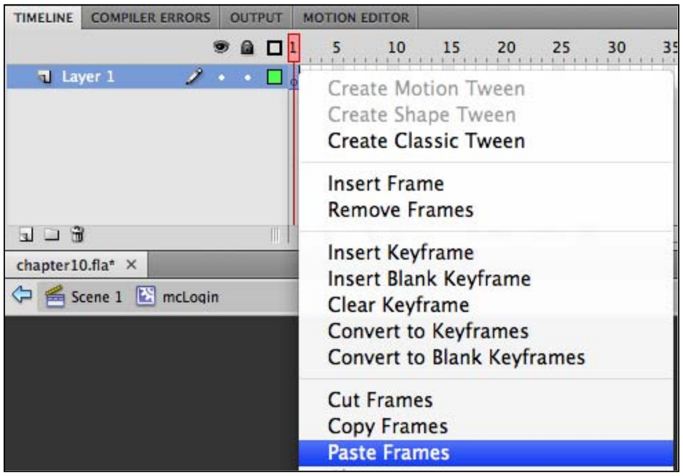
Create New Symbol

Name: mcLogin

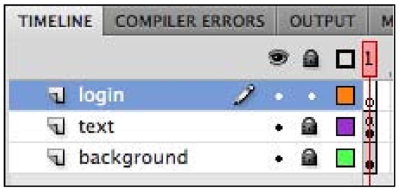
Type: Movie Clip

Folder: Library root

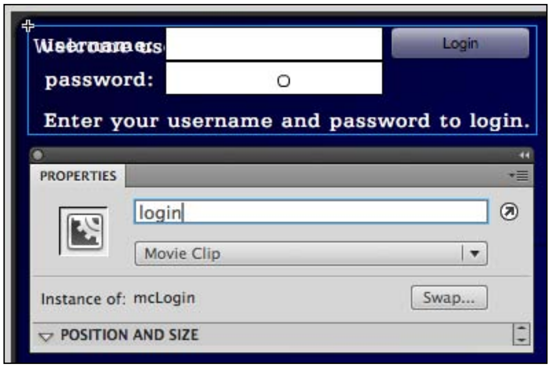
This will create an empty movie clip, where we can right-click on the default key frame and select **Paste Frames** to copy all the layers and objects that we just created into this movie clip.



Now that we have a complete **mcLogin** movie clip, our next task is to move back to the main stage, remove all of the login layers that we created, and then delete the **userpass** movie clip from the **login** layer.



With the empty **login** layer selected, we can now drag and drop our new **mcLogin** movie clip from our **Library** so that it is the only movie clip within this layer. When we are done placing our **mcLogin** movie clip, we make sure that we give it an instance of **login**.



login

Movie Clip

Instance of: mcLogin

Our next task will be to build the ActionScript code that will govern the logic of our user login block so that we can log in to our Drupal web site and start adding data. But first, we need to handle the currently logged in user within our Flash application.

User handling within Flash

In order to handle the current user, we will first need to open up the `main.as` file, where we will build the business logic behind the Flash objects we created in the previous section. Since each of these objects was given an instance name, they can now be referenced within our ActionScript code. Looking at the `main.as` file, our first task will be to instantiate our user login block by hiding the welcome movie clip.

```
// Hide the welcome message.
login.welcome.visible = false;

// Declare our Drupal connection
var drupal:NetConnection = new NetConnection();
drupal.objectEncoding = ObjectEncoding.AMF3;

// Connect to the Drupal gateway
drupal.connect( gateway );
```

Our next task is to show the correct login block depending on whether the user is logged in or not. This can be determined from the user object passed to us after we connect to our Drupal system. If the user is defined and the user ID is not 0, then this means that we are logged in, and should show the welcome block and hide the login block. We will also want to fill out the text within the **username** text field inside the welcome block to be that of the logged-in user. All of this functionality can be placed within its own function called `setUser`. Since, the currently logged-in user is someone who we will need to reference at a later time, we will also need to add a global declaration for the currently logged-in user at the top of the `main.as` file, and then set the value within this function.

```
// The current user.
var user:Object = null;
// Hide the welcome message.
login.welcome.visible = false;
// Declare our Drupal connection
var drupal:NetConnection = new NetConnection();
drupal.objectEncoding = ObjectEncoding.AMF3;
// Connect to the Drupal gateway
drupal.connect( gateway );
...
...
// Called when Drupal returns with a successful connection.
function onConnect( result:Object )
{
   // Set our current user.
   setUser( result );
   trace("We are connected!!!");
   trace("Session Id: " + sessionId);
   // Load our node.
   loadNode( nodeId );
}
// Used to set the current user.
function setUser( result:Object )
{
   // Set our sessionId variable.
   sessionId = result.sessid;
   // Set the global variable.
   user = result.user;
   // Check to see if we are logged in.
   if( user && user.userid > 0 ) {
      // Welcome our user.
      login.welcome.visible = true;
      login.welcome.username.text = user.name;
```

```
            login.userpass.visible = false;
         }
         else {
            // Show the login block.
            login.welcome.visible = false;
            login.userpass.visible = true;
         }
      }
```

Along with the **welcome** and **userpass** movie clips, we will also need to change the text within our login button so that it says `Login` if the user is not logged in and `Logout` otherwise.

```
   // Used to set the current user.
   function setUser( _user:Object )
   {
      // Set our sessionId variable.
      sessionId = result.sessid;

      // Set the global variable.
      user = result.user;
      // Check to see if we are logged in.
      if( user && user.userid > 0 ) {
         // Welcome our user.
         login.submit.label = "Logout";
         login.welcome.visible = true;
         login.welcome.username.text = user.name;
         login.userpass.visible = false;
      }
      else {
         // Show the login block.
         login.submit.label = "Login";
         login.welcome.visible = false;
         login.userpass.visible = true;
      }
   }
```

All of this functionality is used to set and display the Flash objects based on whether the user is logged in or not. However, we must now complete this functionality by hooking up the login button so that we can trigger an event when the user clicks on the button to log in.

Hooking up our login button

In order to log into Drupal, we will need to hook up our login button so it can be used to log the user in or out depending on whether they are logged in or not. We can do this by adding an event handler when this button is clicked.

```
// The current user.
var user:Object = null;

// Hide the welcome message.
login.welcome.visible = false;

// Add a listener when the user presses the Login button.
login.submit.addEventListener( MouseEvent.CLICK, onLogin );
```

We now need to create our stub function that will get called whenever the user clicks this button.

```
// Used to set the current user.
function setUser( result:Object )
{
   // Set our sessionId variable.
   sessionId = result.sessid;
   // Set the global variable.
   user = result.user;
   // Check to see if we are logged in.
   if( user && user.userid > 0 ) {
      // Welcome our user.
      login.submit.label = "Logout";
      login.welcome.visible = true;
      login.welcome.username.text = user.name;
      login.userpass.visible = false;
   }
   else {
      // Show the login block.
      login.submit.label = "Login";
      login.welcome.visible = false;
      login.userpass.visible = true;
   }
}

// Called when the user clicks the login button.
function onLogin( event:MouseEvent )
{
   trace( "Login clicked" );
}
```

At this point, we can now add functionality into the onLogin function so that it will log the user into our Drupal system. But first, we will need to perform some pre-validation on the username and password.

Checking for a username and password

In this section we need to check to make sure that the user provided both their username and password into the text fields, and if they have not, then we will change the text within the status text field to let the user know. In order to make our code very robust, we need to make sure that we remove any unwanted characters from the username and password that could have been added involuntarily by the user such as tabs or spaces. We can do this by performing a search for all of those characters, and replacing them with an empty string. We will use the String replace function, which takes a regular expression as the search parameter. Although the explanation of regular expressions is somewhat beyond the scope of this book, we can use the regular expression /[\t\n\r\f]/ to search for any tabs or new line indicators within the text field. If there are any of these characters within our string, they will be replaced with an empty string.

Our changes should look like the following:

```
// Called when the user clicks the login button.
function onLogin( event:MouseEvent )
{
   // Get the username and password.
   var username:String = login.userpass.username.text;
   var password:String = login.userpass.password.text;

   // Replace unwanted characters.
   username = username.replace(/[\t\n\r\f]/,'');
   password = password.replace(/[\t\n\r\f]/,'');

   // Check for the username and password
   if( username.length && password.length ) {
      trace("Login to Drupal");
   }
   else {
      login.status.text = "username and password required.";
   }
}
```

Now that we have some preliminary checks on the username and password, we can make our call to Drupal to log in the user.

Logging into Drupal

Our next task will be to utilize the User Service that we set up earlier in this chapter to allow us to perform a remote login using Flash. Within the User Services module, there are two different remote functions that we can use to log in and log out of our Drupal system. These two functions are `user.login` and `user.logout` respectively, and depending on whether the user is logged into the system or not, we will need to call one or the other. But first, we will need to set up our responder that will be used to contain our callback functions when the server returns after each remote function is called.

Adding a user responder

We will create a single responder that we will use to handle both the login and logout functionality. This responder will be called `userResponder`, and we can declare it as follows:

```
// Set up our responder with the callbacks.
var responder:Responder = new Responder( onConnect, onError);
// User responder to handle Login and Logout commands.
var userResponder:Responder = new Responder( setUser, onUserError );
// Connect to Drupal
drupal.call("system.connect", responder);
```

Our next step is to create the `onUserError` callback function that the responder calls when Drupal has returned with an error during the login process. This function will be very useful to pass along the error from our Drupal server, so that the user knows what went wrong with their user login. For example, this error handler would be able to tell our user if they accidentally misspelled their username by giving them a message that says "Wrong username and password". This function can be defined as follows:

```
// Used to set the current user.
function setUser( result:Object )
{
   // Set our sessionId variable.
   sessionId = result.sessid;
   // Set the global variable.
   user = result.user;
   // Check to see if we are logged in.
   if( user && user.userid > 0 ) {
      // Welcome our user.
```

```
        login.submit.label = "Logout";
        login.welcome.visible = true;
        login.welcome.username.text = user.name;
        login.userpass.visible = false;
     }
     else {
        // Show the login block.
        login.submit.label = "Login";
        login.welcome.visible = false;
        login.userpass.visible = true;
     }
  }
  // Called when an error occurs during login.
  function onUserError( error:Object )
  {
     // Set the status to what happened.
     login.status.text = error.description;
  }
```

Now that we have our responder in place, our next step is to utilize it by calling the User Service routines to log into our Drupal system.

Logging in

Logging into Drupal can be accomplished by placing a `drupal.call` to the User Service routine when the user clicks on the login button. This can be done by placing the `drupal.call` at the point where both `username` and `password` have been checked within the `onLogin` function. We will also make sure that the user is not logged in before calling this functionality.

```
  // Called when the user clicks the login button.
  function onLogin( event:MouseEvent )
  {
     // Is the user logged out?
     if( user && user.userid == 0 ) {
        // Get the username and password.
        var username:String = login.userpass.username.text;
        var password:String = login.userpass.password.text;
        // Replace unwanted characters.
        username = username.replace(/[\t\n\r\f]/,'');
        password = password.replace(/[\t\n\r\f]/,'');
        // Check for the username and password
        if( username.length && password.length ) {
        drupal.call( "user.login", userResponder, sessionId, username,
        password );
```

```
         }
         else {
            login.status.text = "username and password required.";
         }
      }
   }
```

Now that we have login functionality, this system would not be complete without letting the user log out of the Flash application.

Logging out

Logging out of the Flash application requires us to make the `drupal.call` to `user.logout` when the user is currently logged into the Flash application. Since we are already changing the name of the login button to say `Logout` when the user is logged in, we will utilize the same button handler function (`onLogin`) to handle the case when the user also wishes to log out of the system.

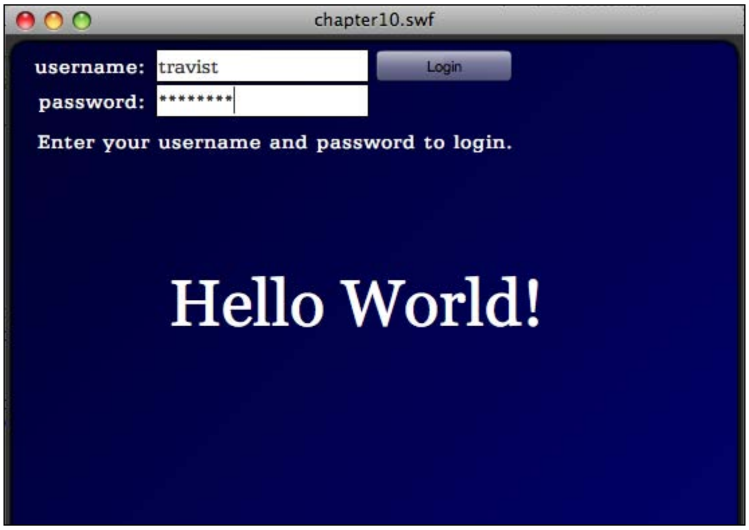
```
   // Called when the user clicks the login button.
   function onLogin( event:MouseEvent )
   {
      // Is the user logged out?
      if( user && user.userid == 0 ) {
         // Get the username and password.
         var username:String = login.userpass.username.text;
         var password:String = login.userpass.password.text;
         // Replace unwanted characters.
         username = username.replace(/[\t\n\r\f]/,'');
         password = password.replace(/[\t\n\r\f]/,'');
         // Check for the username and password
         if( username.length && password.length ) {
            drupal.call( "user.login", userResponder, sessionId,
            username, password );
         }
         else {
             login.status.text = "username and password required.";
         }
      }
      else {
         // Log out of Drupal.
         drupal.call( "user.logout", userResponder, sessionId );
      }
   }
```

We run into a problem, however, when our Drupal site responds to our request. For the login, we are using the `userResponder`, which in turn uses the `setUser` function as the callback function that gets triggered when the server returns with its response. The problem comes in that the response of the `user.login` service is different from the response of the `user.logout` service. Instead of redefining a new responder for the logout functionality, we can still use the `setUser` function as the callback for the logout response, but we will need to make some changes to account for the different response.

The response from `user.logout` is different from `user.login` in that it returns a simple Boolean value notifying the remote application as to whether the logout was successful or not. Because of this, we can determine if the response is from the logout service call by checking the result value for the Boolean type. If it is a Boolean, then we can reset our `userId` to 0, which will place our application in the logged out state.

```
// Used to set the current user.
function setUser( result:Object )
{
   if( result is Boolean ) {
      // Set our userId to 0
      user.userid = 0;
   }
   else {
      // Set our sessionId variable.
      sessionId = result.sessid;
      // Set the global variable.
      user = result.user;
   }
   // Check to see if we are logged in.
   if( user && user.userid > 0 ) {
      // Welcome our user.
      login.submit.label = "Logout";
      login.welcome.visible = true;
      login.welcome.username.text = user.name;
      login.userpass.visible = false;
   }
   else {
      // Show the login block.
      login.submit.label = "Login";
      login.welcome.visible = false;
      login.userpass.visible = true;
   }
}
```

And we are done…We can now run our application and see a fully functional user login system!



username: travist

password: ********

Enter your username and password to login.

Hello World!

Summary

In this chapter we learned how to log into Drupal's user management system from a remote Flash application. Although this may seem like a trivial exercise, it completely opens up the doors to some more exciting possibilities with data management and manipulation from a remote location. It will serve as a necessary segue to the following chapters where user management is required for data manipulation. In the next chapter, we will utilize this user management to learn how to add data to Drupal using Flash. We will also learn how to take advantage of all the security measures that the Services module puts into place by re-enabling the API key functionality that we temporarily disabled in this chapter.

11
Adding Content to Drupal

Now that we have set the foundation for retrieving content from within Drupal, we can focus our efforts on turning that one-way street of data transfer into a two-way street, where we can not only retrieve content, but also add and manipulate that content from within a remote Flash application. Obviously, this may raise some concerns regarding security, which is why we will cover how Drupal allows for this type of remote interaction, while, at the same time, keeping our data safe. In this chapter we will learn how to add content to our remote Drupal web site, from within Flash, by building a custom Flash node editor that will allow us to create, view, and edit nodes within the Drupal system.

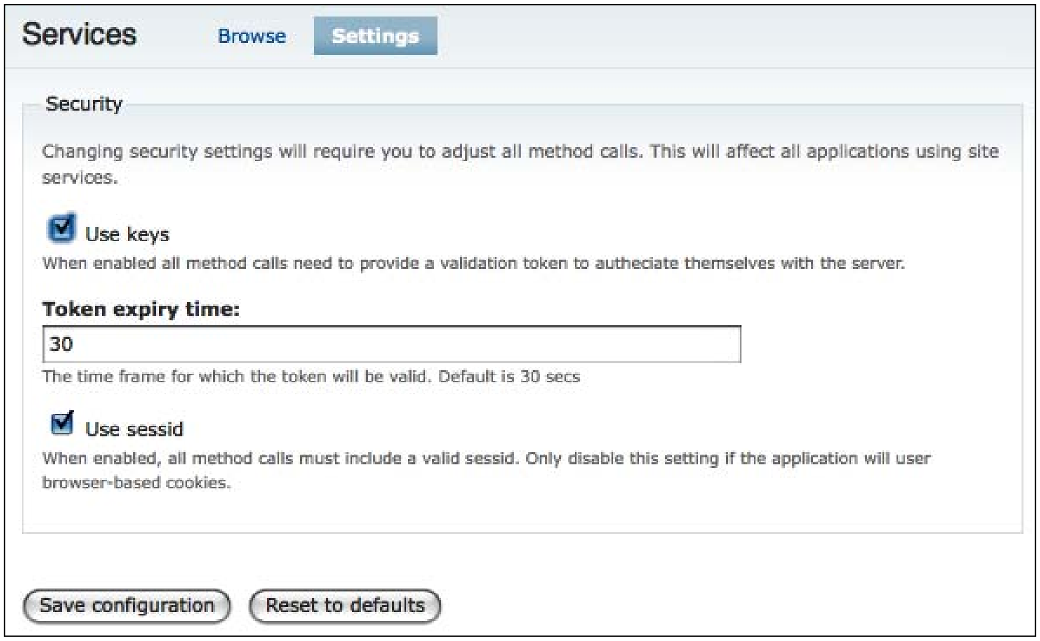
In this chapter, we will cover the following points:

- Learning about Drupal Services Security
- Building a Drupal Service class
- Building a node editor in Flash
- Adding content to Drupal from Flash
- Editing existing content in Drupal from Flash

Drupal Services security

In recent times, security has become crucial for any web site tapped into the treacherous environment that is the World Wide Web. Because of the ever increasing popularity of Drupal, much effort has been placed on ensuring that any person using this incredible content management system does not fall victim to a security breach in their web site. Although it is impossible to plug all the holes, we can put our faith in Drupal that they are covering most bases when it comes to the security of our data. And the Services module is no exception. However, opening up the ability for any external web application to add and manipulate data within our web site is something we will want to approach with caution. Because of this, it is extremely important that we are very familiar with the security measures that the Services module employs to ward off malicious software.

In the previous chapter, we caught a glimpse of these security measures when we opened up the API for the **user.login** web service. Due to the complexities of adding support for these security measures, we temporarily disabled them to illustrate a user login system. However, for this chapter, we will re-enable these security measures and then build a new Drupal Service class to help abstract out all of the complexities when using them for any future Flash applications. To start, we will navigate to **Administer | Services** on our Drupal site, where we will re-enable the API keys within the settings of our Services Administrator by clicking on the **Use keys** checkbox and then clicking on the **Save Configuration** button.



Security

Changing security settings will require you to adjust all method calls. This will affect all applications using site services.

☑ Use keys

When enabled all method calls need to provide a validation token to autheciate themselves with the server.

Token expiry time:

30

The time frame for which the token will be valid. Default is 30 secs

☑ Use sessid

When enabled, all method calls must include a valid sessid. Only disable this setting if the application will user browser-based cookies.

Now that we have the Services API keys enabled, we can explore how the Services module keeps our data safe by looking at what makes up the API key.

The API key

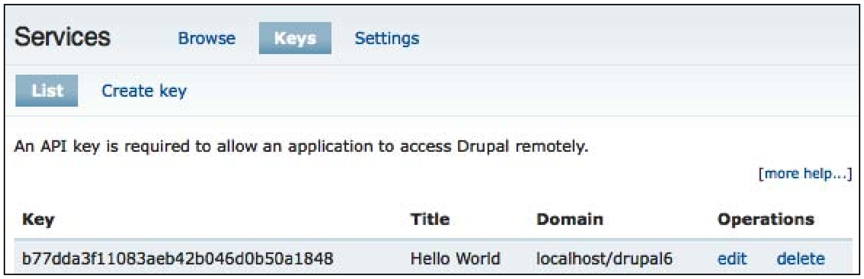
The API key, or **Application Programming Interface** key, is used to help keep our data safe by keeping track of a set of keys for any external applications to use when performing data critical tasks. The Services module uses this unique identifier to check any service requests to make sure that whoever made that call has the right credentials to do whatever he/she is trying to do.

It is much easier to think of the API key by giving an analogy of an underground club that we see in the movies. If we stick true to Hollywood, you would almost always see some cool character, wearing sunglasses in the middle of the night, walk up to a metal door with a small sliding notch window. Then, the eyes of some guard are revealed from behind that sliding window asking, "What's the secret password?" Then, the cool and collected character would respond "abra-cadabra" (or probably something much cooler).

In the nerdy world of web services, this scenario is captured as the Services module asking some remote application what the secret password is…or, in other words, what is the API key? Then, the cool and collected remote application would present the API key and in return gain entry into the night club (metaphorically speaking). In Chapter 2, we briefly covered the use of these API keys and even created one for our Hello World application. One thing that we did not do, however, was go over in great detail what an API key actually is and how it is used to protect our system from malicious software.

API key configuration

To start, we will explore the current API key configuration that we set up within Chapter 2 by navigating to the **Keys** section of our Services Administrator where we should see the following:



An API key is required to allow an application to access Drupal remotely.

Key	Title	Domain	Operations
b77dda3f11083aeb42b046d0b50a1848	Hello World	localhost/drupal6	edit delete

What this is showing is the API key that we created in Chapter 2, which includes the key (shown within the **Key** column), along with the **Title** of the application and the **Domain**. Since we can use the same API key for multiple applications, we can easily edit this key by clicking on the **edit** link within the **Operations** column.

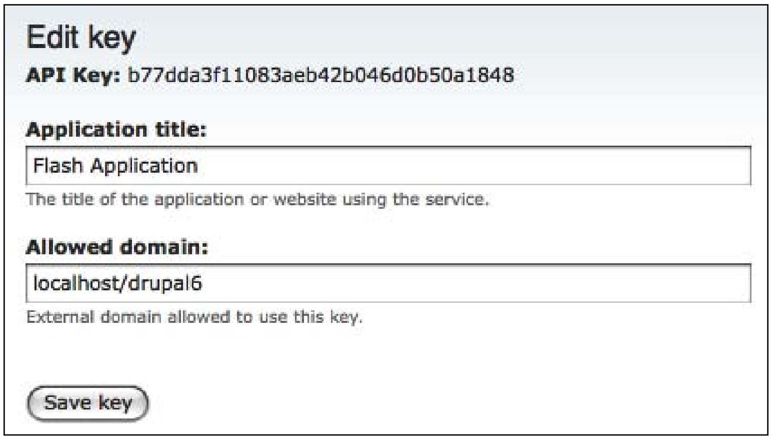
Within this page, we can provide both the **Application title** and the **Allowed domain** for our API key. The application title is the name of the application that will be using the web service API key. This is generally only used to keep track of what each key is used for, and is not necessarily important to get a working system. Since we wish to use it for any Flash application that we create, we can change the name from **Hello World** to simply **Flash Application**. The allowed domain is the important value when creating a new API key.

The allowed domain and crossdomain.xml

The allowed domain is the value that gets populated within the `crossdomain.xml` file of our Drupal site. The `crossdomain.xml` file is a file used by Flash to determine which sites it can communicate with and which sites it cannot. The Services module automatically generates this file using the Drupal menu system. We can see the contents of this simulated file by visiting our domain followed by `crossdomain.xml`, such as `http://localhost/drupal6/crossdomain.xml`. Although we do not have a `crossdomain.xml` file within our Drupal root on our server, the menu system generates this file using the values that we provide in this API key section. Therefore, it is important to provide a value for our allowed domains that will populate this `crossdomain.xml` file with the correct values. It is also important to note that the `http://` should not be included when providing the allowed domain, but any subdirectory that our Drupal root resides in should be included. Following are a list of examples that give the domain name for a web site along with the correct values for the allowed domain values:

Drupal root	**Allowed Domain**
`http://www.mysite.com`	`www.mysite.com`
`http://mysite.com`	`mysite.com`
`http://www.mysite.com/subdir`	`www.mysite.com/subdir`
`http://mysite.com/subdir`	`mysite.com/subdir`
`http://localhost`	`localhost`
`http://localhost/subdir`	`localhost/subdir`

Once we have finished filling out our API key, we should have the following:



Edit key

API Key: b77dda3f11083aeb42b046d0b50a1848

Application title:

Flash Application

The title of the application or website using the service.

Allowed domain:

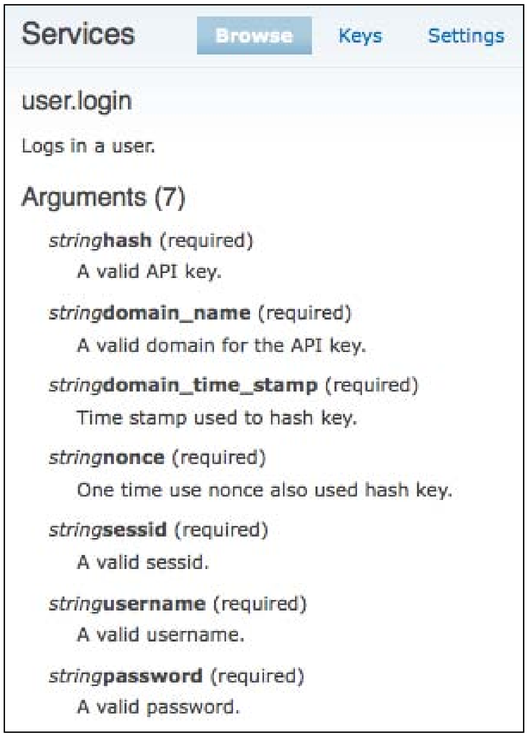
localhost/drupal6

External domain allowed to use this key.

When we are done, we can save our key by clicking on the **Save key** button. After we do this, we should be able to navigate back to our `crossdomain.xml` file, where we should then see our domain added to the list of allowed domains. Now that we have edited our API Key, we can move on to the next section, where we will learn how to use it within a remote Flash application.

How to use the API key

We can see firsthand how the API key works by clicking on the **Browse** link of the Services Administrator, where it lists all of the services available to external applications. Since not all service calls require an API key, we will need to select a service that performs data critical operations on our Drupal data in order to see what the API key does and what is required to implement this security check. We can see this key by selecting the same service that we did in the previous chapter by clicking on the **user.login** link to view the API declaration for that service. We should see something similar to the following:



user.login

Logs in a user.

Arguments (7)

*string***hash** (required)
A valid API key.

*string***domain_name** (required)
A valid domain for the API key.

*string***domain_time_stamp** (required)
Time stamp used to hash key.

*string***nonce** (required)
One time use nonce also used hash key.

*string***sessid** (required)
A valid sessid.

*string***username** (required)
A valid username.

*string***password** (required)
A valid password.

Although we cannot see the API key when viewing any of these arguments, the API key, along with all the other arguments, are used when calculating the hash argument. Each of these arguments is explained as follows:

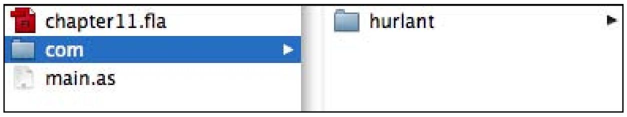
- **hash**—The hash is an encoded string that is made up of the API key along with the **domain_name**, **domain_time_stamp**, and **nonce** arguments. It uses an HMAC encoding to create a hash value of all of these values combined.
- **domain_name**—This is the domain name that should match what is provided as the allowed domain that we just created for the API key.
- **domain_time_stamp**—This is a timestamp of when the external call was made. This is used so that each hash will have a timeout value and only be valid for a certain time interval.
- **nonce**—Also written as N-once, this is a single random string that is only allowed for a single call. The Services module keeps track of all nonce strings for each service call and checks them to make sure that they are never reused. This keeps any malicious software from trying to mimic any service call by using the same security parameters. Since the nonce can only be used once, the second (malicious) call will end up failing.
- **sessid**—The session ID for the current user session. Each user for our web site will get a unique session, and this can be used to validate each user for the privileges that they have on our Drupal site.

Now that we have a clear understanding of what each of these arguments are, we are ready to dive back into Flash where we will build a system that is able to provide all the necessary elements to perform data critical operations.

Building a Drupal service in Flash

We can shift our focus back to Flash, where we will build a reusable system for interfacing with Drupal using the API keys where applicable. To begin this section, we will need to make a copy of our `chapter10` directory and create a new `chapter11` directory with the contents of that folder.

Once we have our new `chapter11` folder created, our task will be to get all of the necessary components needed to generate the hash required from Drupal's API key. Luckily, all of the hard work has already been done thanks to the wonderful open-source development community and the kind folks at `http://crypto.hurlant.com`. The library that we will use is called `as3crypto` and can be downloaded at `http://as3crypto.googlecode.com/files/Crypto.zip`. This library gives us the ability to generate any type of hash, including the HMAC required by the Services module. Once we have the ZIP package downloaded, we will then extract the contents of this package and copy the **com** folder within this package into our `chapter11` directory.



Now that we have the `as3crypto` library in place, we can start to build our new class that we will use to encapsulate all of the special functionality required for the Drupal service interaction.

The DrupalService class

In all the previous chapters we interfaced with Drupal using the simple `NetConnection call` method. With each of these calls to our Drupal system, the API key was not required, and consequently, it was not necessary to build a separate class for interacting with our Drupal web site from a remote location. However, now, we have a special need to make certain service calls that require an added security measure, which should be encapsulated within its own class. Because of this, we will use this section to build a re-usable `DrupalService` class that can be used within any Flash project to interact with Drupal. To start, we will simply need to create a new ActionScript file within our `chapter11` directory called `DrupalService.as` file.

Within our new `DrupalService.as` file, we will want to derive all functionality from the `NetConnection` class so that we can expand on that functionality to include the API key. At this point, our code should look like the following:

```
package
{
   // Import all dependencies
   import flash.net.NetConnection;
   // Declare our DrupalService class
   public class DrupalService extends NetConnection
   {
   }
}
```

At this point, we will want to provide the same functionality as the `NetConnection` class, but build in our Drupal-specific needs into this class. For example, we can keep track of our `sessionId` and user object within this class, as well as build in our connection to Drupal, which sets these parameters:

```
package
{
   // Import all dependencies
   import flash.net.NetConnection;
   import flash.net.Responder;
   // Declare our DrupalService class
   public class DrupalService extends NetConnection
   {
      // The DrupalService constructor.
      public function DrupalService( _gateway:String, _api:String )
      {
         // Save the gateway path and apiKey.
         gateway = _gateway;
         apiKey = _api;
         // Set the object encoding and connect to Drupal.
         objectEncoding = flash.net.ObjectEncoding.AMF3;
         super.connect(gateway);
      }
      // Connect to our Drupal system.
      public function connectToDrupal( _onConnected:Function )
      {
         // Set our callback function.
         onConnected = _onConnected;
         // Call our system.connect service.
         call("system.connect", new Responder(onConnect, onError));
      }
      // Called when we have connected to Drupal
      private function onConnect( result:Object )
      {
         // Set our session Id and user object.
         sessionId = result.sessid;
         user = result.user;
         // Call the callback function.
         onConnected( {user:user, sessid:sessionId} );
      }
      // Called when an error occurs connecting to Drupal.
      private function onError( error:Object )
      {
         trace("An error has occurred!");
      }
      // Store our user and session Id.
      public var user:Object;
      public var sessionId:String="";
      public var gateway:String = "";
      public var apiKey:String = "";
      // Set the onConnected callback function.
      private var onConnected:Function;
   }
}
```

Now that we have our `DrupalService` class working to connect and retrieve information from our Drupal system, we will add to it to allow the additional API key for certain service calls.

Adding the API key to our DrupalService

Since the API key is only required for the services where data manipulation is required, we will need to structure our DrupalService class to either use the API key for some service calls, while at the same time, only use the session ID for others. Because of this, we will need to create our very own serviceCall method within our DrupalService that will handle these custom arguments depending on which service is called. Also, because the declaration of the call method, within the NetConnection class, uses the variable arguments declaration (…), we will need to be clever when we pass on the arguments to the NetConnection call method. Unfortunately, the only way to do this is to create a switch for the maximum amount of arguments that we can expect to be passed to this function, and then call each NetConnection method for that number of arguments. This can be seen as follows:

```
// Connect to our Drupal system.
public function connectToDrupal( _onConnected:Function )
{
   // Set our callback function.
   onConnected = _onConnected;
   // Call our system.connect service.
   serviceCall( "system.connect", onConnect, null );
}
   // Create a service Call method.
   public function serviceCall(command:String, onSuccess:Function,
   onFailed:Function, ... args)
{
   // Use default error handler if none is provided.
   if( onFailed == null ) {
      onFailed = onError;
   }
   // Declare our responder.
   var responder:Responder = new Responder(onSuccess, onFailed);
   // Switch and make the call.
   switch ( args.length ) {
   case 0 :
      call( command, responder );
      break;
   case 1 :
      call( command, responder, args[0] );
      break;
   case 2 :
      call( command, responder, args[0],
                                args[1] );
      break;
   case 3 :
      call( command, responder, args[0],
                                args[1],
                                args[2] );
```

```
         break;
      case 4 :
         call( command, responder, args[0],
                                     args[1],
                                     args[2],
                                     args[3] );
         break;
      case 5 :
         call( command, responder, args[0],
                                     args[1],
                                     args[2],
                                     args[3],
                                     args[4] );
         break;
      case 6 :
         call( command, responder, args[0],
                                     args[1],
                                     args[2],
                                     args[3],
                                     args[4],
                                     args[5] );
         break;
      case 7 :
         call( command, responder, args[0],
                                     args[1],
                                     args[2],
                                     args[3],
                                     args[4],
                                     args[5],
                                     args[6] );
         break;
      case 8 :
         call( command, responder, args[0],
                                     args[1],
                                     args[2],
                                     args[3],
                                     args[4],
                                     args[5],
                                     args[6],
                                     args[7] );
         break;
      default:
         trace("Too many arguments in DrupalService");
         break;
      }
   }
```

Now that we have our own `serviceCall` routine, our next task is to build up the arguments based on which service is called.

Adding arguments to the service call

Our next task will be to determine which methods require an API key and which do not. This can be done using a simple function within our `DrupalService` class that will let us know if the API key is required given the service command. Although there are several functions that require an API key, we will only include the ones that we will use in our application. This function will look as follows:

```
// Determines if we should use an API Key.
private function usesKey( command:String ):Boolean
{
   switch ( command ) {
      case "user.login":
      case "user.logout":
      case "node.save":
         return true;
         break;
   }
   return false;
}
```

We can add a new function that adds the required arguments for each Flash call to our Drupal site. This new function will take the arguments passed to the `serviceCall` method, and then call the `unshift` method on the arguments array to add each of the required arguments. For the time being, we will not populate all of the parameters, but just use this as a shell to add the additional arguments.

```
// Create a service Call method.
public function serviceCall(command:String, onSuccess:Function,
onFailed:Function, ... args):void
{
   // Use default error handler if none is provided.
   if( onFailed == null ) {
      onFailed = onError;
   }
   // Declare our responder.
   var responder:Responder = new Responder(onSuccess, onFailed);
   // Setup our arguments.
   setupArgs( command, args );
   ...
   ...
}
// Setup the arguments.
private function setupArgs( command:String, args:Object )
{
   // Add the session Id as the first argument.
   args.unshift( sessionId );
   // Check to see if we should add the API key.
   if ( usesKey(command) ) {
      var baseURL:String = "";
      var timestamp:String = "";
      var nonce:String = "";
      var hash:String = "";
      args.unshift( nonce );
      args.unshift( timestamp );
      args.unshift( baseURL );
      args.unshift( hash );
   }
}
```

Our next task is to fill in the blanks by providing functions that get the timestamp, nonce, base URL, and hash values. To help simplify this integration, we will tackle each one of these additions individually.

Adding the base URL

To determine the base URL that was used to generate the API key, we can simply extract this value using the gateway string. Since there are an infinite amount of possibilities for the value of this gateway string, we will need to use some clever regular expressions to extract the right string from this gateway. By looking at the typical gateway string, we can then see how to construct the base URL used for our API key by observing the following rules:

- The typical base URL is highlighted from the following gateway string `http://`**`www.mysite.com/drupal6`**`/services/amfphp`
- We will search for `http://` and remove that from the string leaving `www.mysite.com/drupal6/services/amfphp`
- We will then search for `/services/amfphp` and remove that from the gateway leaving the base URL of `www.mysite.com/drupal6`

The code that realizes this set of rules is realized as follows:

```
   // Setup the arguments.
   private function setupArgs( command:String, args:Object )
   {
      // Add the session Id.
      args.unshift( sessionId );
      // Check to see if we should add the API key.
      if ( usesKey(command) ) {
```

```
      var baseURL:String = getBaseURL();
      var timestamp:String = "";
      var nonce:String = "";
      var hash:String = "";
      args.unshift( nonce );
      args.unshift( timestamp );
      args.unshift( baseURL );
      args.unshift( hash );
   }
}
// Get the baseURL.
private function getBaseURL():String
{
   // The regular expression to find "http://"
   var http:RegExp = /^(http[s]?\:[\\\/][\\\/])/;
   // Remove the "http://" from the gateway.
   var baseURL:String = gateway.replace(http, "");
   // Remove "/services/amfphp" from the gateway.
   baseURL = baseURL.replace("/services/amfphp", "");
   // Return the base URL.
   return baseURL;
}
```

Adding the TimeStamp

The timestamp, we can determine relatively easily by using ActionScript's date module, which has the ability to determine the current system time.

```
// Setup the arguments.
private function setupArgs( command:String, args:Object )
{
   // Add the session Id.
   args.unshift( sessionId );
   // Check to see if we should add the API key.
   if ( usesKey(command) ) {
      var baseURL:String = getBaseURL();
      var timestamp:String = getTimeStamp();
      var nonce:String = "";
      var hash:String = "";
      args.unshift( nonce );
      args.unshift( timestamp );
      args.unshift( baseURL );
      args.unshift( hash );
   }
}
```

```
...
// Get the timestamp.
private function getTimeStamp():String
{
   // Get the current date/time
   var now:Date = new Date();
    // Get the time as a timestamp.
   var nowTime:Number=Math.floor((now.getTime() / 1000));
   // Return it as a string.
   return int(nowTime).toString();
}
```

Adding the nonce

The nonce is simply a randomly generated 10 character string. However, the one problem with ActionScript 3.0 is that there is currently no standard method to create a random string. Because of this, we will just need to get creative by creating a string with all the alphabet characters, and then constructing a new string by randomly selecting characters from that alphabet to create a random string of characters. This can be seen as follows:

```
// Setup the arguments.
private function setupArgs( command:String, args:Object )
{
   // Add the session Id.
   args.unshift( sessionId );
   // Check to see if we should add the API key.
   if ( usesKey(command) ) {
      var baseURL:String = getBaseURL();
      var timestamp:String = getTimeStamp();
      var nonce:String = getNonce();
      var hash:String = "";
      args.unshift( nonce );
      args.unshift( timestamp );
      args.unshift( baseURL );
      args.unshift( hash );
   }
}
...
// Returns the nonce, or 10 character random string.
private function getNonce():String
{
   // Create an alphabet string with all the alphabet.
   var alphabet:String = "abcdefghijkmnopqrstuvwxyz";
   alphabet += "ABCDEFGHJKLMNPQRSTUVWXYZ23456789";
   // Store the length.
```

```
      var len:int = alphabet.length - 1;
      var randString:String = '';
      // Iterate 10 times for each character.
      for (var i:int = 0; i < 10; i++) {
         // Append a random character.
         randString += alphabet.charAt(rand(len));
      }
      // Return our random string.
      return randString;
   }
   // Generate a random number with a max range.
   private static function rand(max:Number):Number
   {
      return Math.round(Math.random() * (max - 1));
   }
```

Adding a hash

Our last step to creating our security checks is to take all three of the previous values and mash them together to create a hash. For this piece, we will need to bring in our as3crypto libraries that we should already have in place. The hash simply consists of the timestamp, domain, nonce, and the Services command all combined and then scrambled using an SHA256 encoded HMAC with the API key as the algorithm seed. We will also need to make sure to include all the necessary encoding libraries to perform this magic. This can be seen as follows:

```
package
{
   // Import all dependencies
   import flash.net.NetConnection;
   import flash.net.Responder;
   import flash.utils.ByteArray;
   import com.hurlant.crypto.hash.SHA256;
   import com.hurlant.crypto.hash.HMAC;
   import com.hurlant.util.Hex;
...
...
// Setup the arguments.
private function setupArgs( command:String, args:Object )
{
   // Add the session Id.
   args.unshift( sessionId );
   // Check to see if we should add the API key.
   if ( usesKey(command) ) {
      var baseURL:String = getBaseURL();
      var timestamp:String = getTimeStamp();
      var nonce:String = getNonce();
      var hash:String = getHash(timestamp,baseURL,nonce,command);
      args.unshift( nonce );
      args.unshift( timestamp );
      args.unshift( baseURL );
      args.unshift( hash );
   }
}
...
// Get the hash of all values.
private function getHash( timestamp:String, domain:String, nonce:
String, method:String ):String
{
   // Combine all values into an input string.
   var input:String = timestamp + ";";
   input += domain + ";";
   input += nonce + ";";
   input += method;
   // Create an SHA256 encoded HMAC object.
   var hmac:HMAC = new HMAC( new SHA256() );
   // Convert the API Key into a ByteArray.
   var kdata:ByteArray = Hex.toArray(Hex.fromString(apiKey));
   // Convert our input string into a ByteArray.
   var data:ByteArray = Hex.toArray(Hex.fromString(input));
   // Compute our hash.
   var currentResult:ByteArray = hmac.compute(kdata,data);
   // Return our hash in string form.
   return Hex.fromArray(currentResult);
}
```

At this point, we can make any service call to Drupal and not have to worry about providing the right credentials for that connection. And because all this functionality is encapsulated within a single `DrupalService` class, we can use this for other Drupal-Flash projects where the API key is necessary to perform data critical tasks.

Our next task is to open up our `main.as` file and make the necessary changes to incorporate our new `DrupalService` component.

Adding DrupalService functionality to main.as

In this section, we will start by opening up our `main.as` file, where we will replace all of our previous `NetConnection` functionality with our newly created `DrupalService` class. This will consist of performing the following tasks:

- Changing the `drupal` variable to a `DrupalService` variable
- Adding the `apiKey` and passing that to the new `drupal` variable

- Deleting all `Responders`, and then adding those callback functions to the `serviceCall` methods they are used for
- Removing all `sessionId` handling and instead using the `DrupalService` to keep track of this value
- Changing `drupal.connect` to `drupal.connectToDrupal`

When we are done making these changes, our new `main.as` file should look as follows, which we should be happy to see is less complicated from the previous version (an indication that we have just done a good thing).

```
// Declare our variables
var baseURL:String = "http://localhost/drupalbook";
var gateway:String = baseURL + "/services/amfphp"
var nodeId:Number = root.loaderInfo.parameters.node;
var apiKey:String = "b77dda3f11083aeb42b046d0b50a1848";

// The current user.
var user:Object = null;

// Hide the welcome message.
login.welcome.visible = false;

// Add a listener when the user presses the Login button.
login.submit.addEventListener( MouseEvent.CLICK, onLogin );

// Declare our Drupal connection
var drupal:DrupalService = new DrupalService( gateway, apiKey );

// Connect to Drupal
drupal.connectToDrupal( onConnect );

// Loads a Drupal node.
function loadNode( nid:Number )
{
   // Call the Drupal service to load the node.
   drupal.serviceCall( "node.get", onNodeLoad, null, nid );
}

// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Print out the node title.
   title.text = node.title;
}

// Called when Drupal returns with a successful connection.
function onConnect( result:Object )
{
   // Set our current user.
   setUser( result );
```

```
      trace("We are connected!!!");
      trace("Session Id: " + drupal.sessionId);
      // Load our node.
      loadNode( nodeId );
   }
   // Used to set the current user.
   function setUser( result:Object )
   {
      if( result is Boolean ) {
         // Set our userId to 0
         user.userid = 0;
      }
      else {
         // Set our sessionId variable.
         drupal.sessionId = result.sessid;
         // Set the global variable.
         user = result.user;
      }
      // Check to see if we are logged in.
      if( user && user.userid > 0 ) {
         // Welcome our user.
         login.submit.label = "Logout";
         login.welcome.visible = true;
         login.welcome.username.text = user.name;
         login.userpass.visible = false;
      }
      else {
         // Show the login block.
         login.submit.label = "Login";
         login.welcome.visible = false;
         login.userpass.visible = true;
      }
   }
   // Called when an error occurs during login.
   function onUserError( error:Object )
   {
      // Set the status to what happened.
      login.status.text = error.description;
   }
   // Called when the user clicks the login button.
   function onLogin( event:MouseEvent )
   {
```

```
   // Is the user logged out?
   if( user && user.userid == 0 ) {
      // Get the username and password.
      var username:String = login.userpass.username.text;
      var password:String = login.userpass.password.text;

      // Replace unwanted characters.
      username = username.replace(/[\t\n\r\f]/,'');
      password = password.replace(/[\t\n\r\f]/,'');

      // Check for the username and password
      if( username.length && password.length ) {
         drupal.serviceCall( "user.login", setUser, onUserError,
           username, password );
      }
      else {
         login.status.text = "username and password required.";
      }
   }
   else {
      // Log out of Drupal.
      drupal.serviceCall( "user.logout", setUser, onUserError );
   }
}

// Called when an error occurs connecting to Drupal.
function onError( error:Object )
{
   for each (var item in error) {
      trace(item);
   }
}
```

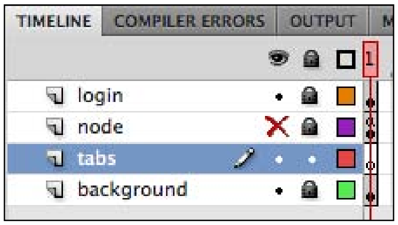
At this point, we are at about the same spot that we left off from the previous chapter, except that one major thing is different…we are handling the API Key transparently within our new `DrupalService` class. By abstracting out this complex functionality into its own class, we have clearly simplified our interface to Drupal by calling the same API routine for all method calls to Drupal. We are then leaving the hard work to our new `DrupalService` class, which determines on its own if more arguments are needed and then supplies them accordingly. We are now ready to start building a node editor within Flash.

Building a node editor in Flash

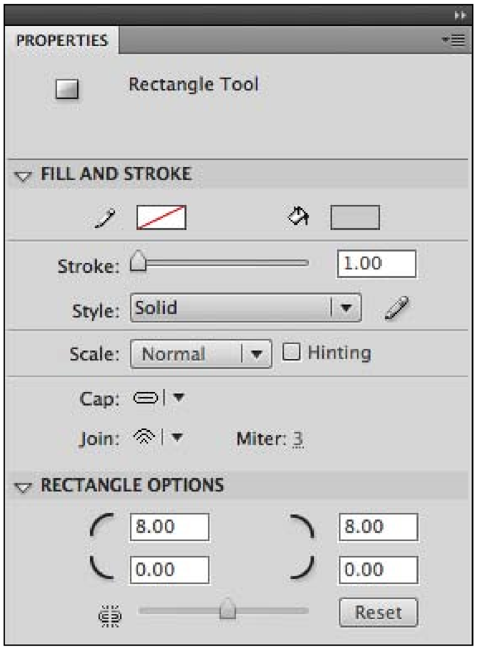
Before we can create our business logic for adding content to our Drupal installation, we will construct our node editor that we will use to view, edit, and add new content to our Drupal web site. Our goal here is to make this editor as easy to use as possible and make it obvious which operation we are performing (viewing, adding, or editing). We can achieve this goal by creating an editor that resembles Drupal's node editor, where there are tabs at the top of the content that allow us to perform specific operations on that content. We can start this off by opening up our `chapter11.fla` project file, where we will make some cosmetic changes to our old Hello World application. For starters, we will create our new tab system that will have our **view**, **edit**, and **add** tabs.

Creating view, edit, and add tabs

Before we begin this section, we will direct our attention to the timeline, where we will rename the current **title** layer to **node**, and then create a new layer called **tabs** that we will use to add all of our tabs. Once we are done with this, we will then hide our **node** layer so that we can have a blank slate to work on when creating our new tabs.



We are now ready to create the background that will be used for each **view**, **edit**, and **new** tab. The easiest way to do this is to use the Rectangle tool and then draw a rounded top rectangle by specifying the following in the Properties panel:



PROPERTIES

Rectangle Tool

FILL AND STROKE

Stroke: 1.00

Style: Solid

Scale: Normal □ Hinting

Cap:

Join: Miter: 3

RECTANGLE OPTIONS

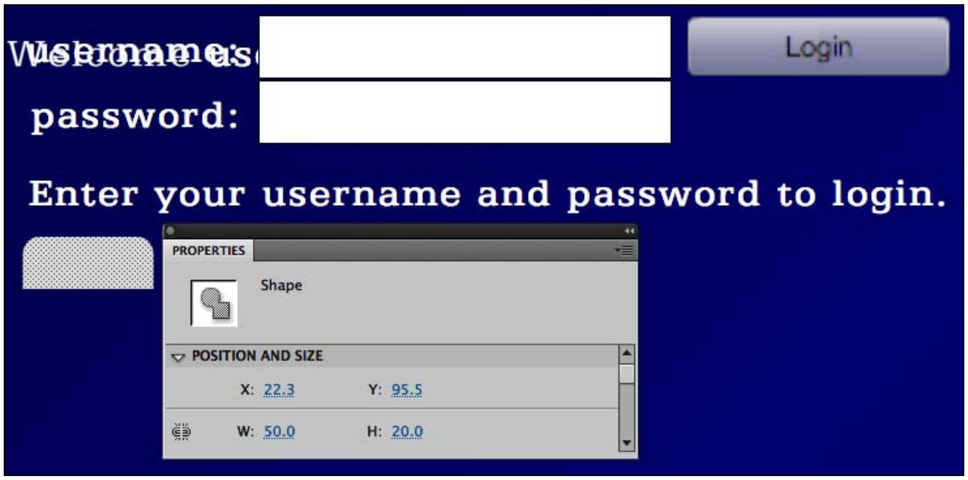
8.00 8.00

0.00 0.00

Reset

> In order to draw a rounded top rectangle, we will need to break the lock on the rounded corner settings so that it does not apply to all the corners. To do this, simply click on the 🔗 symbol in the Rectangle Options. This will allow us to provide a value for only the top corners.

We can then draw a single, rounded top rectangle that is about 50 pixels wide and about 20 pixels tall, like the following illustrates.



PROPERTIES

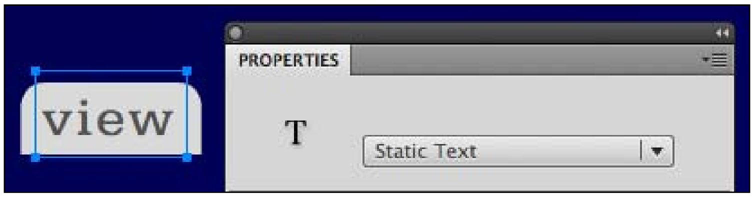
Shape

POSITION AND SIZE

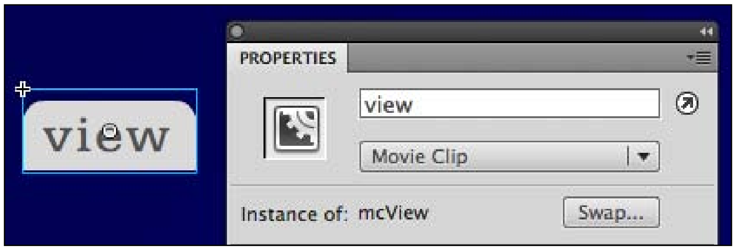
X: 22.3 Y: 95.5

W: 50.0 H: 20.0

Our next task will be to create a **Static Text** field on top of this shape that will be used as the label for this tab. For now, we can just use the label for the first tab, which will be called view.



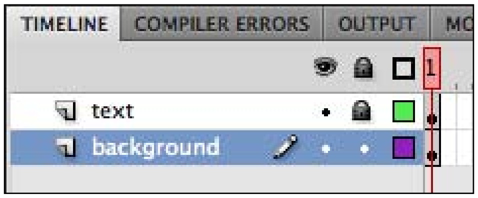
Finally, we can create a new **MovieClip** from both the rectangle and the static text field by selecting both of these objects and selecting **Modify** | **Convert to Symbol** from the menu. We will call this movie clip **mcView**, and then give it an instance name of **view**.



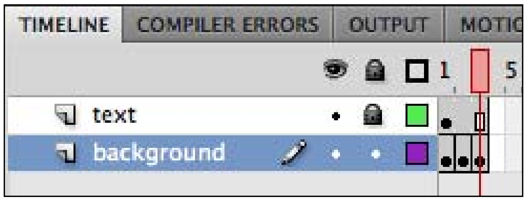
We can edit this movie clip by double-clicking on it, where we will add some timeline labels for the **normal**, **hover**, and **selected** states.

Adding normal, hover, and selected states

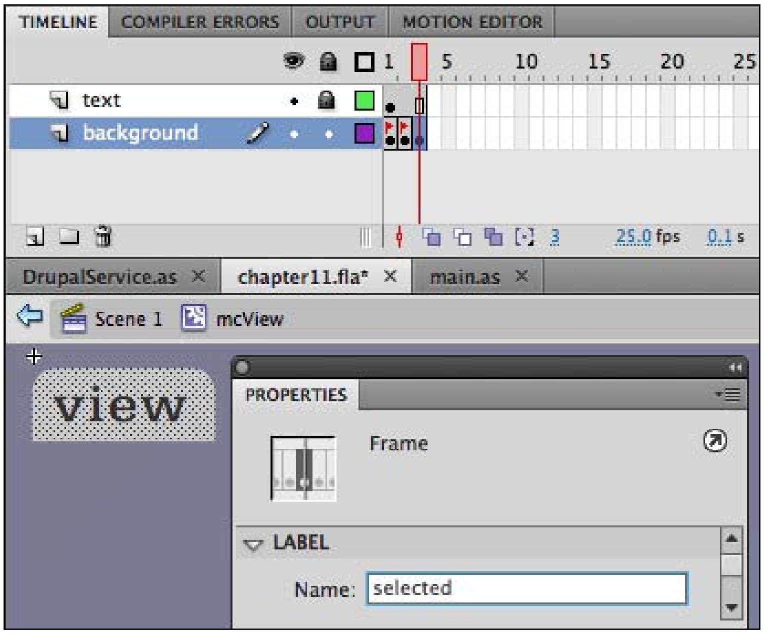
With our new view tab, MovieClip, open, we will need to break apart the text and the background into two separate layers. We will place the text on a layer called **text**, and the background on a layer called **background**, and then lock the **text** layer.



Once this is done, we will add three new key frames to the **background** layer's timeline (by pressing *F6*), and then extend the **text** layer (by pressing *F5*).



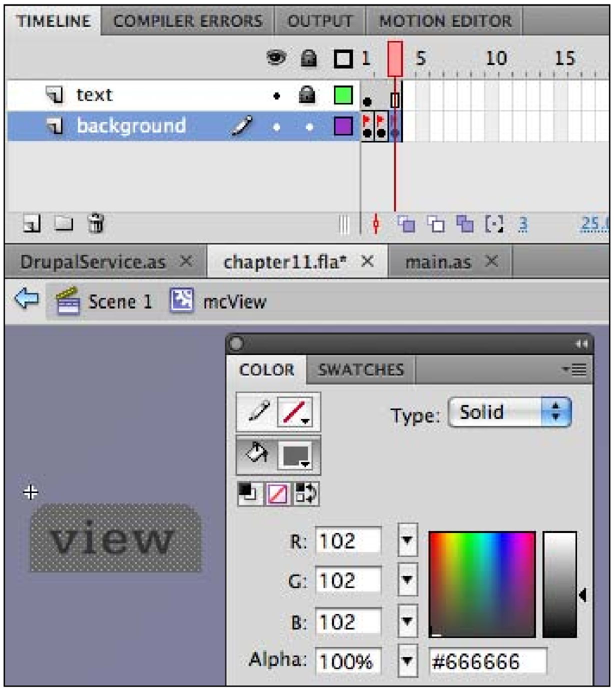
For each of these key frames, we will give them a label of **normal**, **hover**, and **selected.** We can do this by clicking on the key frames within the backround layer, and then typing in the name of that key frame in the Properties panel.



Name: selected

Our last task is to change the background color for the tab for each of these key frames so that we can distinguish them for each state. The following background fill colors are recommended:

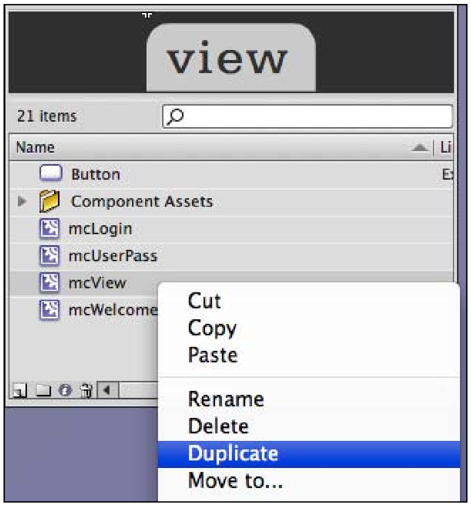
- **normal**—0xCCCCCC
- **hover**—0xFFFFFF
- **selected**—0x666666



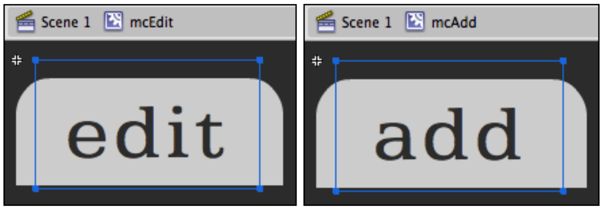
Now that we have done the **view** tab, we can duplicate the **mcView** movie clip three times to create our **edit** and **add** links.

Duplicating the mcView for the edit and add tabs

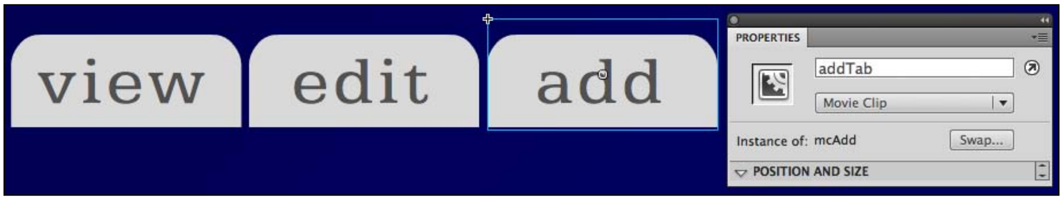
To duplicate our view tab, we will navigate back to our stage movie clip by clicking on the **Scene 1** link in the breadcrumb trail above the stage area. Now that we are back on the root stage, we can then click on the **Library** link and locate our **mcView** movie clip. We will then right-click on **mcView** and select the **Duplicate** drop-down item.



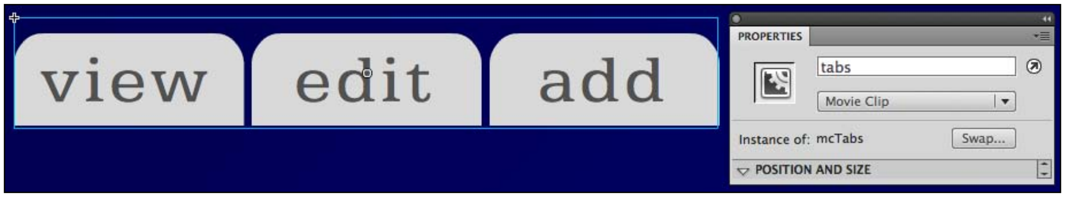
We will need to use this method to create a **mcEdit** and a **mcAdd** movie clip to our library. Now that we have two new duplicate movie clips from the **mcView** movie clip, we can edit each one of them by double-clicking on them within the library and then changing the text within each of them to say **edit** and **add** instead of **view**.



And now moving back to the stage, we can add our new **add** and **edit** tabs so that they are aligned next to the **view** tab as follows. We will need to make sure to give each of these tabs an instance name of **view**, **edit**, and **addTab** (since **add** is a reserved word in Flash) respectively.



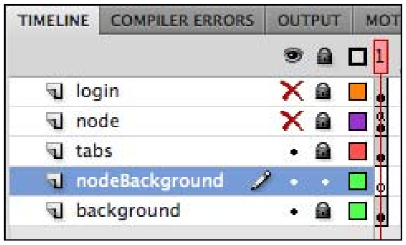
After we have our three different tabs in place, our next task will be to select all three of these tabs and turn them into a single movie clip by clicking on **Modify | Convert to Symbol** from the Flash menu. We will call our new movie clip **mcTabs** and then give it an instance name of **tabs**.



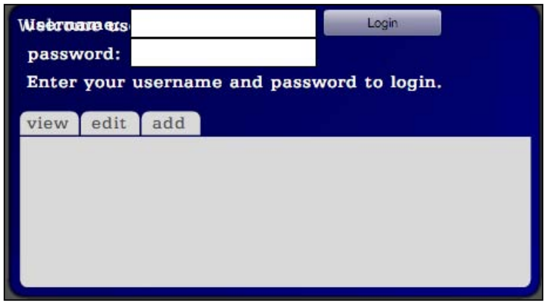
Once we have done with this, we can lock our **tabs** layer and add a background to our node to give it some unity.

Adding a background to our node

By adding a background to our node, we visually tie the tabs that we created with the content that they represent. This can be easily done by creating a new layer above the **background** layer called **nodeBackground**.



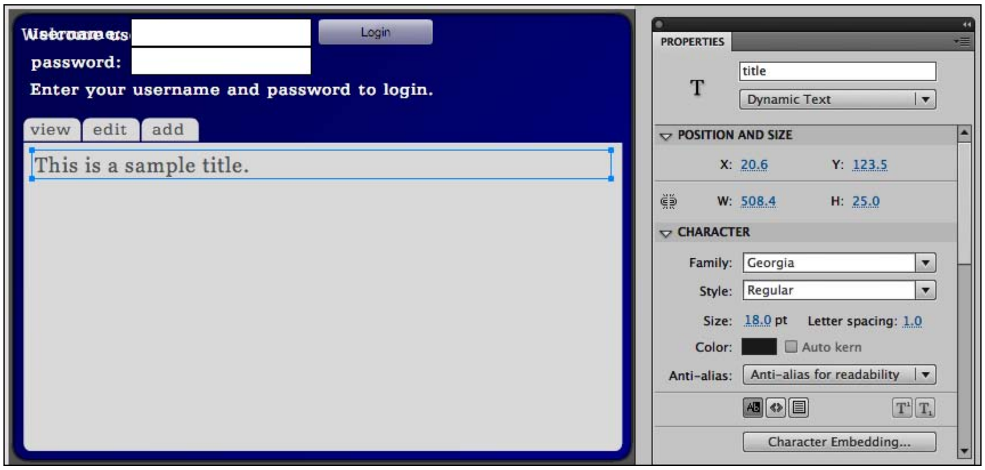
Then, by using the Rectangle tool ▭, we will create a rounded bottom rectangle whose color matches the normal state of the tabs. We should also leave a small gap between the tabs and this background to give it some depth. This should look like the following:



We can lock the **nodeBackground** layer, and then move onto changing our current node view so that it fits within our new node layout.

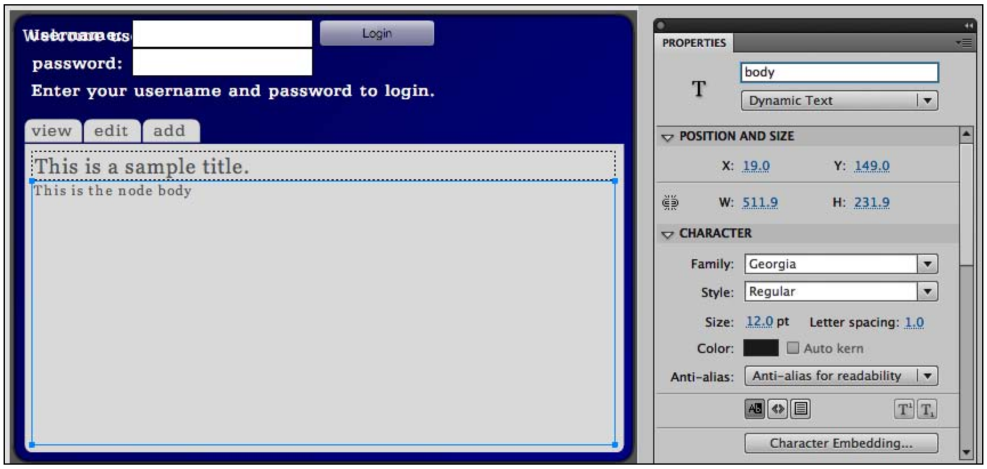
Changing the node view

With the **nodeBackground** layer locked, we will unlock and reveal our **node** layer to expose our current title text field. Our first task will be to resize the **title** text field so that it fits our new node editor region, as well as reducing the font size so that we can have more room for the node body.



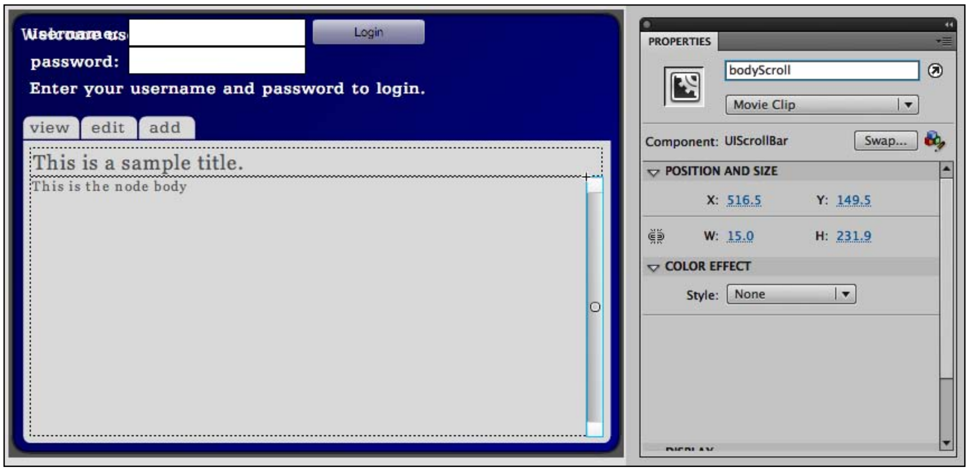
title
Dynamic Text
X: 20.6 Y: 123.5
W: 508.4 H: 25.0
Family: Georgia
Style: Regular
Size: 18.0 pt Letter spacing: 1.0
Color: □ Auto kern
Anti-alias: Anti-alias for readability

Our next task is to add a new **Dynamic Text** field that we will use as the body of our node. We will give it an instance name of **body**, and use a smaller font size, so that we can fit a lot of content within our node editor.



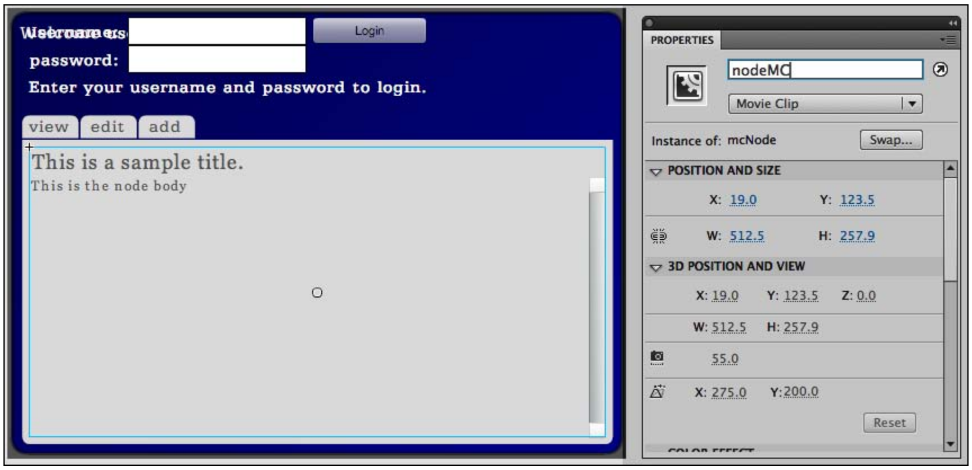
body
Dynamic Text
X: 19.0 Y: 149.0
W: 511.9 H: 231.9
Family: Georgia
Style: Regular
Size: 12.0 pt Letter spacing: 1.0
Color: □ Auto kern
Anti-alias: Anti-alias for readability

And since we will want to place as much text as we would like within the body, we can follow the steps provided in Chapter 3 to add a new **UIScrollBar** to our body field, and then give that scrollbar an instance name of **bodyScroll**.



bodyScroll
Movie Clip
Component: UIScrollBar
X: 516.5 Y: 149.5
W: 15.0 H: 231.9
Style: None

Our final task is to select all of the node components and place them in their own Movie Clip so that it is easier to manage within ActionScript. We will call this movie clip **mcNode** and then give it an instance name of **nodeMC**.

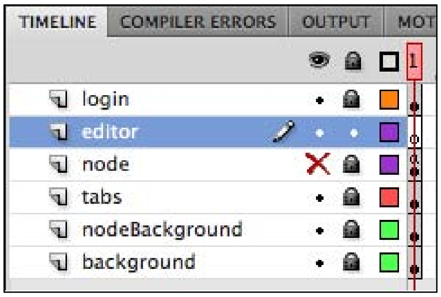


nodeMC
Movie Clip
Instance of: mcNode
X: 19.0 Y: 123.5
W: 512.5 H: 257.9
X: 19.0 Y: 123.5 Z: 0.0
W: 512.5 H: 257.9
55.0
X: 275.0 Y:200.0

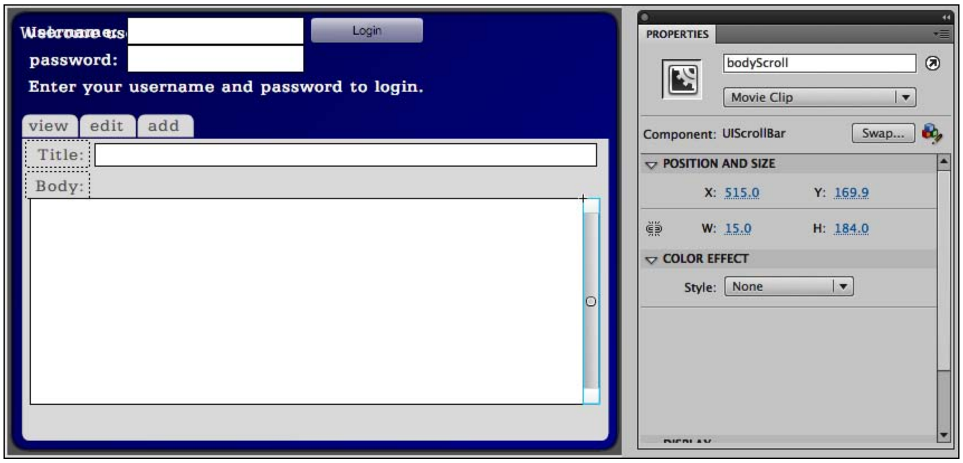
We can now move on to creating our node edit form in Flash.

Creating a node edit form

Now that we have our node view taken care of, our next task is to create our node edit form so that we can add new, and modify existing, Drupal content. We will start this by locking and hiding the **node** layer, and then creating a new layer in our timeline called **editor**.



Within our **editor** layer, we will start by adding two new input text fields (like we did in the previous chapter with the user login) on our stage. We will use these input fields for the **Title** and **Body** of our new node, and give them both an instance name of **title** and **body** respectively. Also, while we are at it, we will probably want to add some static text fields to indicate which input fields are which, and again add a UIScrollBar to the body field called **bodyScroll**. This should look like the following:



bodyScroll

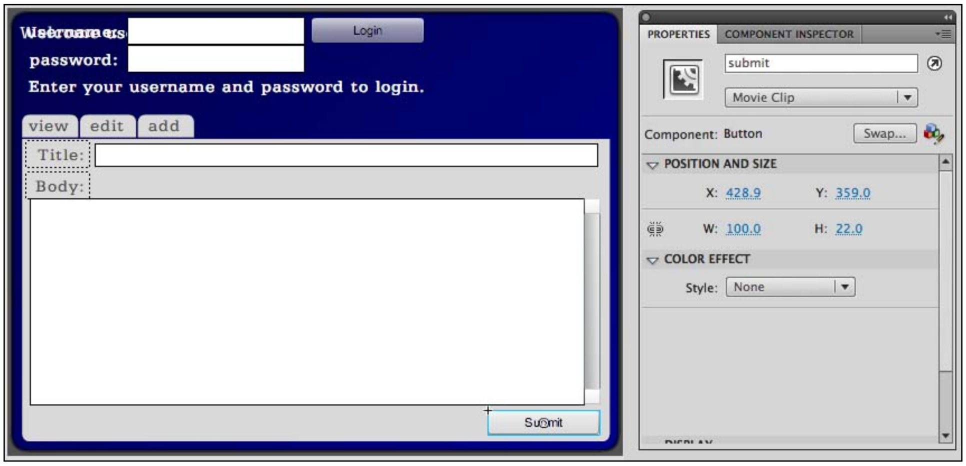
Component: UIScrollBar

X: 515.0 Y: 169.9

W: 15.0 H: 184.0

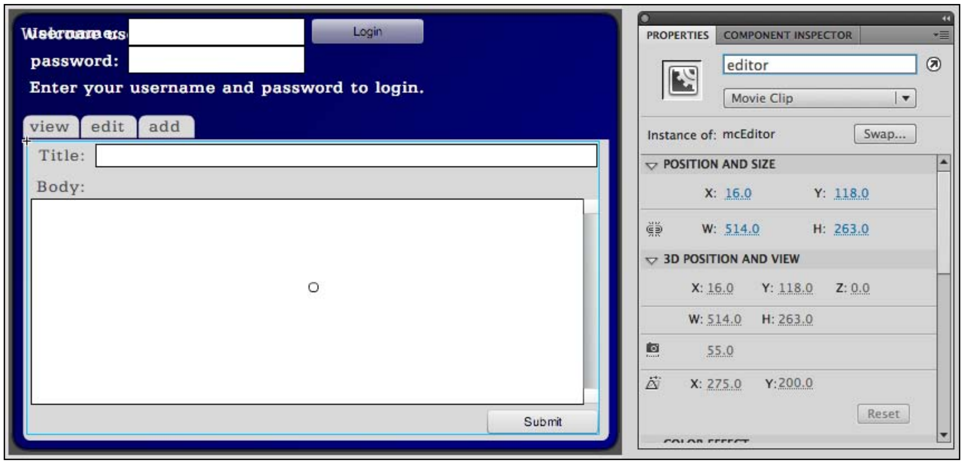
Style: None

Our next task will be to create a Submit button that we will use to submit this new node information to our Drupal web site. We can use the same method that we used for the Login button from the previous chapter to create a new button that says **Submit**, and then give it an instance name of **submit**.



submit
Movie Clip
Component: Button
POSITION AND SIZE
X: 428.9 Y: 359.0
W: 100.0 H: 22.0
COLOR EFFECT
Style: None

Now that we have our node edit fields in place, our next task is to select all of these elements and turn them into a single movie clip by clicking on the **Modify | Convert to Symbol** menu item. We will then give our new movie clip a name of **mcEditor** and an instance name of **editor**.



editor
Movie Clip
Instance of: mcEditor
POSITION AND SIZE
X: 16.0 Y: 118.0
W: 514.0 H: 263.0
3D POSITION AND VIEW
X: 16.0 Y: 118.0 Z: 0.0
W: 514.0 H: 263.0
55.0
X: 275.0 Y:200.0

Our next task will be to modify our `main.as` file to account for these changes and add the business logic required to add content to our Drupal site.

Adding content to Drupal from Flash

Now that we have our new node editor ready to go in Flash, we are ready to add new data to our Drupal web site using this Flash application. We will start by opening up our main.as file, where we will hook up all the business logic that will govern our new node submit form.

Adding tab functionality

Because we added our tabs to our node editor, we will need to create the business logic that will hide and show certain elements when these tabs are clicked. But first, we will need to add event handlers to each tab so that we can handle each state (**normal**, **hover**, and **selected**). We can add the most flexibility by iterating through all of the children within the tabs movie clip and then adding all the handlers we need on all tabs within our tabs movie clip. By doing this, we can easily add new tabs without having to modify our main.as file every time.

```
// The current user.
var user:Object = null;
// Instantiate our tabs.
var i:Number = tabs.numChildren;
while( i-- ) {
   var tab:MovieClip = (tabs.getChildAt(i) as MovieClip);
   tab.gotoAndStop("normal");
   tab.addEventListener( MouseEvent.MOUSE_OVER, onTabHover );
   tab.addEventListener( MouseEvent.MOUSE_OUT, onTabNormal );
   tab.addEventListener( MouseEvent.MOUSE_DOWN, onTabNormal );
   tab.addEventListener( MouseEvent.MOUSE_UP, onTabSelect );
}
// Called when a tab is hovered over.
function onTabHover( event:MouseEvent )
{
}
// Called when a tab goes back to normal.
function onTabNormal( event:MouseEvent )
{
}
// Called when a tab is selected.
function onTabSelect( event:MouseEvent )
{
}
```

At this point, we just need a good way to keep track of the selected tab, and then change the state of the other tabs to the state defined by each mouse event. With a little finessing, we can perform this logic in the following way:

```
// Instantiate our tabs.
var i:Number = tabs.numChildren;
while( i-- ) {
   var tab:MovieClip = (tabs.getChildAt(i) as MovieClip);
   tab.gotoAndStop("normal");
   tab.addEventListener( MouseEvent.MOUSE_OVER, onTabHover );
   tab.addEventListener( MouseEvent.MOUSE_OUT, onTabNormal );
   tab.addEventListener( MouseEvent.MOUSE_DOWN, onTabNormal );
   tab.addEventListener( MouseEvent.MOUSE_UP, onTabSelect );
}
// Set the view tab as the selected tab.
var selectedTab:MovieClip = (tabs.getChildByName("view") as
MovieClip);
selectTab( selectedTab );
// Called when a tab is hovered over.
function onTabHover( event:MouseEvent )
{
   // If we are not the selected tab, then change state.
   if( event.target != selectedTab ) {
      event.target.gotoAndStop( "hover" );
   }
}
// Called when a tab goes back to normal.
function onTabNormal( event:MouseEvent )
{
   // If we are not the selected tab, then change state.
   if( event.target != selectedTab ) {
      event.target.gotoAndStop( "normal" );
   }
}
// Called when a tab is selected.
function onTabSelect( event:MouseEvent )
{
   // Select this tab.
   selectTab(event.target as MovieClip);
}
// Selects a new tab.
function selectTab( newTab:MovieClip )
{
   // Make the previous tab go to normal.
   selectedTab.gotoAndStop("normal");
```

```
        // Set the new selected tab.
        selectedTab = newTab;
        // Go to the selected state.
        selectedTab.gotoAndStop( "selected" );
    }
```

Our last objective is to hide and show the **nodeMC** and editor movie clips depending on which tab is selected. We can do this pretty easily by setting each of these movie clips as visible depending on whether or not the name of the tab that was selected was the **vew** tab.

```
    // Selects a new tab.
    function selectTab( newTab:MovieClip )
    {
        // Make the previous tab go to normal.
        selectedTab.gotoAndStop("normal");
        // Set the new selected tab.
        selectedTab = newTab;
        // Go to the selected state.
        selectedTab.gotoAndStop( "selected" );
        // Hide or show the editor or node view.
        nodeMC.visible = (selectedTab.name == "view");
        editor.visible = (selectedTab.name != "view");
    }
```

And now, we should have the tab functionality of our node editor, and we can move onto the fun stuff of saving and editing nodes using Flash.

Saving a node from Flash

Our first task in this section will be to change the `onNodeLoad` function, where we will add the title and body of our node to our Flash `nodeMC` movie clip. Since the body of the node is in HTML form, we will also need to use the `htmlText` when showing the text within our text field.

```
    // Called when Drupal returns with our node information.
    function onNodeLoad( node:Object )
    {
        // Set the string of our body and title fields.
        nodeMC.title.text = node.title;
        nodeMC.body.htmlText = node.body;
        // Update the node body scroll bar.
        nodeMC.bodyScroll.update();
    }
```

Our next task will be to hook up the functionality of our **Submit** button by adding some event handlers when this button is clicked. We will also want to verify that they have provided the Input title and body fields before we do anything spectacular in this function. If they do not provide these fields when they click on the **Submit** button, we can inform the user using the **status** text field that this is an error. On the other hand, if they do provide the title and body, we will temporarily just add some trace statements to verify that the input fields are being referenced correctly.

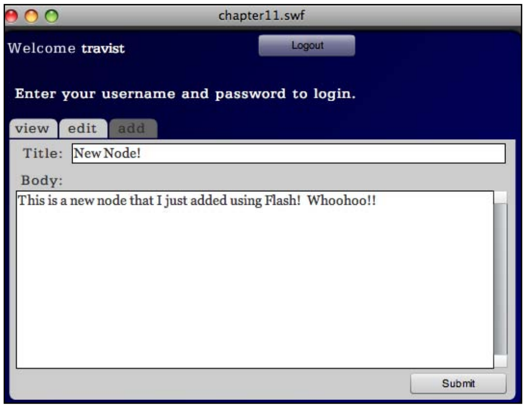
```
// Hide the welcome message.
login.welcome.visible = false;

// Add a listener when the user presses the Login button.
login.submit.addEventListener( MouseEvent.CLICK, onLogin );

// Add a listener when the user presses the Submit button.
editor.submit.addEventListener( MouseEvent.CLICK, onNodeSubmit );
...
...
// Called when the user presses the submit button.
function onNodeSubmit( event:MouseEvent )
{
   // Get the title and body text
   var titleText:String = editor.title.text;
   var bodyText:String = editor.body.text;

   // Replace unwanted characters.
   titleText = titleText.replace(/[\t\n\r\f]/,'');
   bodyText = bodyText.replace(/[\t\n\r\f]/,'');

   // Check to make sure they provide the title and body.
   if( titleText.length && bodyText.length ) {
      trace( titleText );
      trace( bodyText );
   }
   else {
      login.status.text = "You must provide a title and body.";
   }
}
```

We can replace the trace statements that we entered above with a real call to our Drupal Node Service. In order to create a new node in Drupal, we will need to construct a Drupal node object within Flash, and then pass that node as an argument to our Drupal web site. The node object will need three different parameters when it is created: the **title**, **body**, and the **node type**. The title and body are the text strings that were entered in our input **title** and **body** text fields. The node type is simply the type of node that we would like to create with our Service call. We can create a **Page** node type by simply providing "page" as the value for this variable.

After we make our call to Drupal to save our node using the **node.save** service, we will also need to remember to handle the return from Drupal, which will simply be the ID of the node that was just created. We can then take that new node ID and make a call to **node.get** to get all of the node information and populate our title and body fields to show our new node. As for the error, we will simply display what happened in our login status text to let the user know what happened.

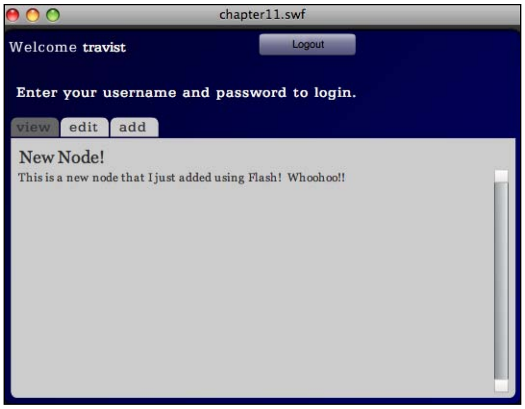
```
// Called when the user presses the submit button.
function onNodeSubmit( event:MouseEvent )
{
   // Get the title and body text
   var titleText:String = editor.title.text;
   var bodyText:String = editor.body.text;

   // Replace unwanted characters.
   titleText = titleText.replace(/[\t\n\r\f]/,'');
   bodyText = bodyText.replace(/[\t\n\r\f]/,'');

   // Check to make sure they provide the title and body.
   if( titleText.length && bodyText.length ) {
      // Create a new node object.
      var newNode:Object = new Object;
      newNode.type = "page";
      newNode.title = titleText;
      newNode.body = bodyText;
      drupal.serviceCall("node.save", onNodeCreate, onNodeError,
      newNode);
   }
   else {
      login.status.text = "You must provide a title and body.";
   }
}

// Called when a new node has been created.
function onNodeError( error:Object )
{
   // Set the status to what happened.
   login.status.text = error.faultString;
}

// Called when a new node has been created.
function onNodeCreate( nodeId:Number )
{
   // Make a call to get the node we just created.
   drupal.serviceCall( "node.get", onNodeLoad, null, nodeId );
}
```

We can run our Flash application and test this out by logging into our Flash application, and then adding a title and body to our input fields and clicking on Submit. After we click on the **Submit** button, we should be able to click on our **view** tab and be able to see the following:



Title: New Node!

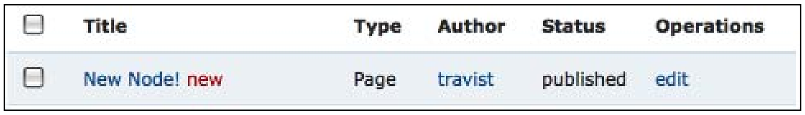
Body: This is a new node that I just added using Flash! Whoohoo!!



New Node!

This is a new node that I just added using Flash! Whoohoo!!

We can verify that this node was actually created by making our way to the Drupal Administrator and navigating to **Administer | Content**, which will show us all the content in our system, including our new node!



□	Title	Type	Author	Status	Operations
□	New Node! new	Page	travist	published	edit

Now that we have successfully added content to our Drupal web site, our next task will be to add functionality to edit existing content.

Editing existing content in Drupal

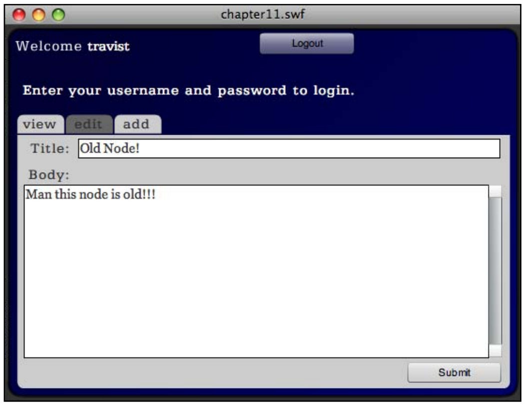
At this point, we simply need to modify our existing functionality so that it is capable of editing nodes as well as adding new ones. Fortunately, we can use the same editor movie clip to do both editing and adding, but will need to populate the fields within this editor, depending on which tab is selected. We will need to add some functionality within our `selectTab` function, that will populate the title and body fields with node information if we are within the **view** tab, and clear them otherwise. To do this, we will need to store the value of the current node that has been populated within the view tab, and then use that stored copy to add text to our title and body fields of the editor for the **edit** tab.

```
// The current user.
var user:Object = null;
var currentNode:Object = null;
...
...
// Selects a new tab.
function selectTab( newTab:MovieClip )
{
   // Make the previous tab go to normal.
   selectedTab.gotoAndStop("normal");
   // Set the new selected tab.
   selectedTab = newTab;
   // Go to the selected state.
   selectedTab.gotoAndStop( "selected" );
   // Hide or show the editor or node view.
   nodeMC.visible = (selectedTab.name == "view");
   editor.visible = (selectedTab.name != "view");
   // Populate the editor title and body.
   if( selectedTab.name == "edit" && currentNode ) {
      editor.title.text = currentNode.title;
      editor.body.htmlText = currentNode.body;
   }
   else {
      editor.title.text = "";
      editor.body.htmlText = "";
   }
   // Update the editor scroll bar.
   editor.bodyScroll.update();
}
...
...
// Called when Drupal returns with our node information.
function onNodeLoad( node:Object )
{
   // Store the current node.
   currentNode = node;
   // Set the string of our body and title fields.
   nodeMC.title.text = node.title;
   nodeMC.body.htmlText = node.body;
}
```

The last change that we need to make is to specify the node ID of the current node to our `node.save` function if we are within the edit tab. By giving our service the node ID, it tells the node service that we would like to update the node rather than create a new one. This can be done within the `nodeSubmit` function as follows:

```
// Called when the user presses the submit button.
function onNodeSubmit( event:MouseEvent )
{
   // Get the title and body text
   var titleText:String = editor.title.text;
   var bodyText:String = editor.body.text;
   // Replace unwanted characters.
   titleText = titleText.replace(/[\t\n\r\f]/,'');
   bodyText = bodyText.replace(/[\t\n\r\f]/,'');
   // Check to make sure they provide the title and body.
   if( titleText.length && bodyText.length ) {
      // Create a new node object.
      var newNode:Object = new Object;
      if( selectedTab.name == "edit" ) {
         newNode.nid = currentNode.nid;
      }
      newNode.type = "page";
      newNode.title = titleText;
      newNode.body = bodyText;
      drupal.serviceCall("node.save", onNodeCreate, onNodeError,
      newNode);
   }
   else {
      login.status.text = "You must provide a title and body.";
   }
}
```

So once we run this, we should be able to pat ourselves on the back, because we should have a fully functional node editor!

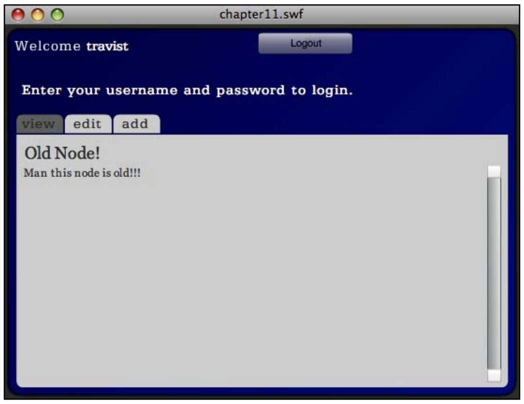

chapter11.swf

Welcome travist

Enter your username and password to login.

Title: Old Node!

Body: Man this node is old!!!


chapter11.swf

Welcome travist

Enter your username and password to login.

Old Node!

Man this node is old!!!

Summary

It is amazing to consider the amount of possibilities that present themselves when we build our remote applications that not only read content but are also capable of adding to and manipulating that content on the fly. This chapter really serves as an enabler for many great things and creative solutions to some really tough problems that many developers may face when trying to add and manipulate Drupal content from within a remote Flash application. It also gives those developers the ability to perform such operations while at the same time preserving data integrity against malicious software. To summarize, we learned the following points in this chapter:

- Data security handling by the the Services module
- Building a Drupal node editor in Flash
- Encapsulating all Drupal communication within a `DrupalService` class
- Building our DrupalService to utilize security measures that the Services module uses to keep our data safe
- Adding data to Drupal using our node editor
- Editing that data using the same node editor

In the final chapter we will bring all lessons learned in this book together for one final project. We will build a five star voter by building our own custom Drupal service that hooks into the very popular Voting API module. This new service will not only add votes, but retrieve them as well, to populate our Flash driven five star voting widget.

12
Build a Drupal Five-star Voter in Flash

Although this is the final chapter in this book, you can expect anything but the typical "sum-it-all-up" theme from this chapter. Of course, we will utilize all the core lessons learned from previous chapters and add onto that knowledge by building our very own custom Drupal Service that will allow customized interaction between Flash and Drupal. This information is vital for any web site that wishes to create its own server tasks and make those services available for external web applications. Once we learn how to build a custom service, we can then apply that knowledge to interface with any subsystem within Drupal (voting, shopping carts, and so on).

Instead of discussing the necessary techniques involved to perform such feats, we will walk through the evolution of a real Flash application that requires its own custom service routine. We will do this by building our very own Flash-driven five-star voter that has the ability to rate content within any Drupal web site. I know that there is already a fantastic module called FiveStar within Drupal that gives us a five-star voting mechanism, but by building our own voter within Flash we can utilize some cool animations within our voter that we would not be able to do within a JavaScript voting mechanism. This alone can open your imagination to some innovative and appealing voting mechanisms for Drupal that will set your web site apart from all the rest. In this chapter, I will give you the foundation to the basics behind building a Flash voter for Drupal. Hopefully, this foundation will spark your imagination to create some truly remarkable voting system that will get everyone's attention, but I will leave that part up to you. In this chapter, we will cover the following points:

- Building a custom Voting Service for Drupal
- Building a five-star voter in Flash
- Creating a Voter class in ActionScript

Building a custom Voting Service for Drupal

Our first task in this chapter will be to create a new **API** (**Application Programming Interface**) for our external application to get and set votes for any piece of content in our Drupal web site. This will require us to create a new module, where we will use the existing Services hook structure to add to the existing services in our system. In order to accomplish this, we first need a general overview on how to create a new module, as well as having a brief overview on how the module hook system works, so that we can add our custom functionality without modifying any core functionality.

It is also important to note that the Voting Service module that we will create in this section has already been built by me and can be downloaded from the Dash Media Player module at `http://www.drupal.org/project/dashplayer`. Although this module is already available to the public, it is important to walk through this process since this information can be easily used to construct new services for any custom application. To start, we will need to tell Drupal about our new module, and we can do that using the module `info` file.

The module info file

The `info` file is used as a registration file so that Drupal can determine whether the contents of the directory, in which it resides, make up the functionality of a module. Drupal, in turn, will then add that module to its list of modules available in the Drupal administrator section. We will start this by opening up a new text document using our favorite text editor. With our empty file open, we will give our module a name and description as follows:

```
name = Voting Service
description = Provides a voting service.
```

We will now provide a location to show our new module within the modules list. Since we already have a Services module section, we can add our new module along with all the rest of the services by adding the following line.

```
name = Voting Service
description = Provides a voting service.
package = "Services - services"
```

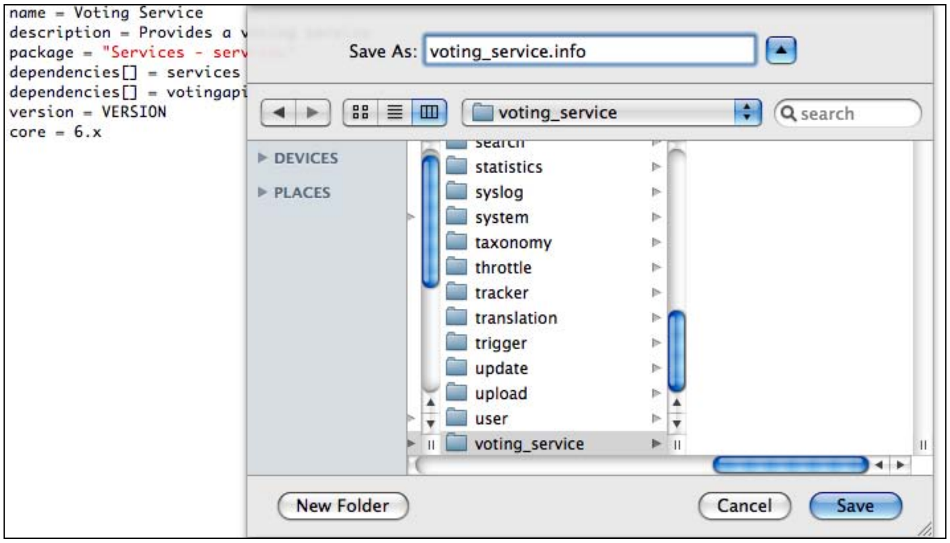
Our next task is to add any dependency that this module has on other modules. What this means is that we do not want our users to install this module without them already having another module that we depend on installed as well. For our Voting Service module, our users are required to have both the **Services** and **Voting API** modules installed, before they are allowed to install our **Voting Service** module (makes sense, huh?). To illustrate this in our `info` file, we will now add the following lines:

```
name = Voting Service
description = Provides a voting service.
package = "Services - services"
dependencies[] = services
dependencies[] = votingapi
```

Finally, we will need to give Drupal some version information so that it can determine if it is compatible with the Drupal installation in which it resides. This can be done by adding the following lines of code to our text editor:

```
name = Voting Service
description = Provides a voting service.
package = "Services - services"
dependencies[] = services
dependencies[] = votingapi
version = VERSION
core = 6.x
```

We are now done with our voting service info file. We can save it in a location where our Drupal server will be able to recognize it. If you are using a localhost server, then this simply requires saving this file as **voting_service.info** within a **voting_service** folder within your site's **modules** directory as follows:



```
name = Voting Service
description = Provides a v
package = "Services - serv
dependencies[] = services
dependencies[] = votingapi
version = VERSION
core = 6.x
```

Save As: voting_service.info

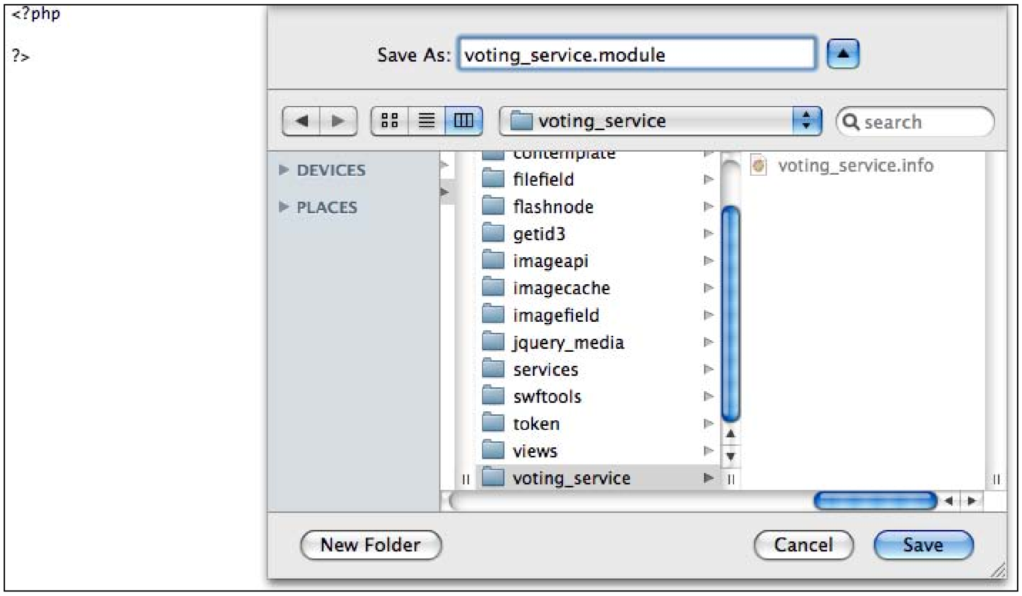
Now that we have our module `info` file in place, our next task is to create the module file that will give us a new voting service.

Voting Service module

With our text editor still open, we will now need to create a new text file, where we will build our Voting Service module. Our first goal should be to simply get the module to install in our Drupal web site and only create the functionality necessary to view it in the modules list, and then install it. By doing this, we can make incremental changes and then test each of those changes. To start with, we need to place our PHP tags within this file since it is technically going to be parsed with a PHP parser.

```
<?php
?>
```

With that simple code in our file, let's now save our file right next to the **voter_service.info** file, and then name this file as **voter_service.module**.



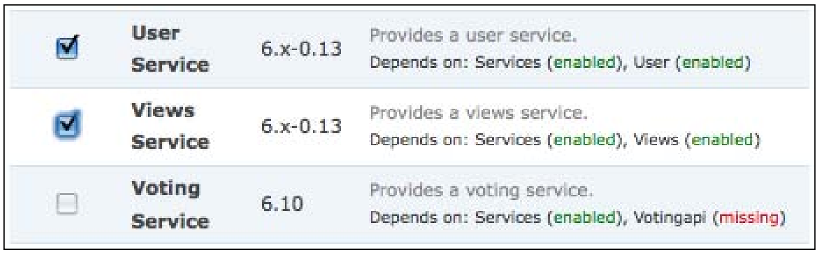
```
<?php

?>
```

Save As: voting_service.module

Surprisingly, this is all we really need to make our module show up within the Drupal modules section, and we can now install our module.

Installing the Voting Service module

We will now navigate to our **Drupal Administrator | Modules** section, where we should see our new module showing up in the modules list.



☑	**User Service**	6.x-0.13	Provides a user service. Depends on: Services (enabled), User (enabled)
☑	**Views Service**	6.x-0.13	Provides a views service. Depends on: Services (enabled), Views (enabled)
□	**Voting Service**	6.10	Provides a voting service. Depends on: Services (enabled), Votingapi (missing)

At this point, we can see that we need to install the Voting API module, which can be found at `http://www.drupal.org/project/votingapi`, before we can enable our new Voting Service. Once we install the Voting API module, we can install our new Voting Service module by clicking on the checkbox next to the Voting Service module, and then clicking on the button at the bottom of the screen that says **Save Configuration**. Congratulations, you have just created your first module…granted, it doesn't do anything, but it is at least recognized by Drupal and ready to rock and roll! Now, we just need to make it actually do something.

Building a Custom Service

In order to add functionality to our new Voting Service, we first need to learn a little bit about the "hook" system, which allows any contributed module to add or extend default functionality. A hook simply works by using naming conventions to find and execute pieces of code by matching the name of the function within that external module. For example, if I have a custom module that wishes to hook into Drupal's node API functionality, I can simply declare a new function called `{module name}_nodeapi`, where I would replace the `{module_name}` with the name of my custom module. So, if our Voting Service module needed to hook into the Node API to edit and modify any node data, I would simply declare a new function called `voting_service_nodeapi`, and then add whatever functionality I need within this function. This concept is the foundation of any module for the Drupal core as well as any of its contributed modules.

The Services module comes with its very own hook system that allows any custom module to literally "hook" into its functionality and add themselves to its list of routines to publish for web services. What this means for us is that we can define a series of functions that can be called from a remote application to perform any custom task. Keep in mind that this can be anything that we can do in PHP, so the amount of possibilities is limitless.

Registering external web services using hook_service

We will start our voting_service.module by first declaring our hook into the Services module, where we can define the routines we would like to publish to external applications. The hook that we are interested in is called hook_service. This hook is simply a registration mechanism, where we will pass back an array of all of the external methods we would like to publish to the world. But for now, we can simply add the prototype for this function to our module file as follows:

```
<?php
/**
 * Implementation of hook_service()
 */
function voting_service_service() {
 return array();
}
?>
```

Our next task is to return an array of any external web services that we would like to use for our Voting Service module. As mentioned before, these are simply functions that can be called from any web application (with the correct credentials) to perform any imaginable server-side task. Within this array is another array, which contains all functions that we would like to "register" to the outside world. We will start by simply declaring the name of the functions that we would like to create along with a description of each method. But first, we need to figure out which methods we will need in order to have a fully functional Flash voter. Obviously, we will need a web service for getting, setting, and deleting votes.

- vote.getVote
- vote.setVote
- vote.deleteVote

But we will also need a way for any user to get his or her own vote. This is needed so that we can show the voters if they have voted for that piece of content or not, and then give them the number of stars that they previously voted. This will require another web service routine.

- vote.getUserVote

We can register these functions by declaring a series of arrays that describe each function we would like to register. But first, we will simply give the names and descriptions of each using the tags #method and #help for each function that we would like to create.

```
<?php
/**
 * Implementation of hook_service()
 */
function voting_service_service() {
  return array(
    // vote.get
    array(
      '#method'   => 'vote.getVote',
      '#help'     => t('Returns a vote.')
    ),

    // vote.getUserVote
    array(
      '#method'   => 'vote.getUserVote',
      '#help'     => t('Returns a users vote.')
    ),
    // vote.set
    array(
      '#method'   => 'vote.setVote',
      '#help'     => t('Submit a new vote.')
    ),
    // vote.delete
    array(
      '#method'   => 'vote.deleteVote',
      '#help'     => t('Delete a vote.')
    )
  );
}
?>
```

Defining web service callback functions

Now that we have our functions defined, we are ready to trigger real functions when each of these web services are called from a remote location. We can do this by first creating a set of functions for each of the services that we defined above. For now, we will just stub these functions out with no functionality.

```
/**
 * Implementation of hook_service()
 */
function voting_service_service() {
  ...
  ...
}

/**
 * Returns a specified node.
 */
function voting_service_get_vote() {
}
/**
 * Gets a vote for the specified user.
 */
function voting_service_get_user_vote() {
}
/**
 * Sets a vote
 */
function voting_service_set_vote() {
}
/**
 * Deletes a vote.
 */
function voting_service_delete_vote() {
}
```

We now need to register these functions as callbacks by assigning them to each of the web services we declared in the `hook_service` function. This can be done using the simple `#callback` tag within each web service registration array as follows:

```
/**
 * Implementation of hook_service()
 */
function voting_service_service() {
  return array(
    // vote.get
    array(
      '#method'   => 'vote.getVote',
      '#callback' => 'voting_service_get_vote',
      '#help'     => t('Returns a vote.')
   ),
    // vote.getUserVote
    array(
      '#method'   => 'vote.getUserVote',
      '#callback' => 'voting_service_get_user_vote',
      '#help'     => t('Returns a vote.')
   ),
    // vote.set
    array(
      '#method'   => 'vote.setVote',
      '#callback' => 'voting_service_set_vote',
      '#help'     => t('Submit a new vote.')
   ),
    // vote.delete
    array(
      '#method'   => 'vote.deleteVote',
      '#callback' => 'voting_service_delete_vote',
      '#help'     => t('Delete a vote.')
   ),
  );
}
```

We are now ready to define the arguments, returns, and logic for each service call.

Adding arguments and voting logic using Voting API

In order to complete the callback function declarations, we first need to define the arguments and return values for each one of these callback routines. Since we are using the Voting API as our interface for all voting functionality, we will need to basically provide the same function requirements that the Voting API requires when getting, adding, and deleting votes. We can pass these parameters along to the Voting API and then allow it to do all of its voting magic behind the scenes. But first, we will need to learn a little bit about how the Voting API works.

The Voting API module

The Voting API is a fantastic module that allows votes to be cast on any form of content (not just nodes). Not only that, it also allows someone to vote on a specific criteria for each piece of content. As an example, we can build a voter for a Recipe node that allows our users to vote on the "difficulty", "taste", and "value" for each recipe node. This functionality is huge, and is something that the FiveStar module does not allow. Because of this, we will need to build our API functions so that they take the following parameters when locating a specific vote within our system:

- content_type—This is the type of content we are voting on. This can be anything from nodes, comments, users, and so on.
- content_id—This is the ID for the piece of content that we are voting on. For nodes, this is the Node ID (nid); for users, this is the User ID (uid); for comments, this is the Comment ID (cid), and so on.

- content_tag—This is the criteria that we are voting on for each individual piece of content. As described above, this can be any tag, such as "difficulty", "taste", "value", and so on.

Given this information, we can construct our routines so that they can first locate the specific vote within the Voting API and then perform that operation on that particular vote. Each of these parameters can be passed to the Voting API routines as the criteria for finding the vote before the operation is performed (with the only exception being the addition of votes). Also, for the returned parameters, we can use the Voting API to return the modified vote after the operation has been performed. This functionality will look like the following within our callback functions:

```
/**
 * Returns a specified node.
 */
function voting_service_get_vote($content_type, $content_id, $tag =
"vote") {
   // Setup the vote criteria.
   $criteria['content_type'] = $content_type;
   $criteria['content_id'] = $content_id;
   $criteria['tag'] = $tag;
   $criteria['value_type'] = 'percent';

   // Select the vote.
   $votes = votingapi_select_votes($criteria);

   if($votes) {
      return $votes[0];
   }
   else
   {
      return array('type'=> $content_type, 'tag' => $tag,
      'value' => 0);
   }
}

/**
 * Gets a vote for the specified user.
 */
function voting_service_get_user_vote($content_type, $content_id, $tag
= "vote") {
   // Get the current user.
   global $user;

   // If the user is logged in.
   if( $user->uid )
   {
        // Set up our vote criteria.
      $criteria['content_type'] = $content_type;
      $criteria['content_id'] = $content_id;
      $criteria['tag'] = $tag;
      $criteria['uid'] = $user->uid;
      $criteria['value_type'] = 'percent';
      // Select the given vote.
      $votes = votingapi_select_votes($criteria);
   }
   if($votes) {
      return $votes[0];
   }
   else
   {
      return array('type'=> $content_type, 'tag' => $tag, 'value'
      =>0);
   }
}
/**
 * Sets a vote
 */
function voting_service_set_vote($content_type, $content_id, $vote_
value, $tag = "vote") {
   // Setup the new vote.
   $vote['content_type'] = $content_type;
   $vote['content_id'] = $content_id;
   $vote['value'] = $vote_value;
   $vote['tag'] = $tag;
   // Set the vote.
   votingapi_set_votes( $vote );
   // Recalculate and return the result.
   votingapi_recalculate_results($content_type, $content_id, TRUE);
   return voting_service_get_vote( $content_type, $content_id,
   $tag );
}
/**
 * Deletes a vote.
 */
function voting_service_delete_vote($content_type, $content_id, $tag =
"vote") {
   // Get the logged in user.
   global $user;
   // Setup the vote criteria.
   $criteria['content_type'] = $content_type;
   $criteria['content_type'] = $content_id;
   $criteria['tag'] = $tag;
   $criteria['uid'] = $user->uid;
   $criteria['value_type'] = 'percent';

   // Select the vote.
   $votes = votingapi_select_results($criteria);

   if($votes) {
      // If the vote exists, then delete it.
      votingapi_delete_vote(array('vote_id' => $votes[0]->vote_id));
      votingapi_recalculate_results($content_type, $content_id,
      TRUE);
   }

   // Return the result.
   return voting_service_get_vote( $content_type, $content_id, $tag );
}
```

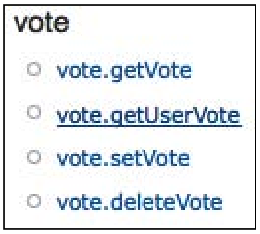
Our next task is to declare each of these arguments within our registration array in the `hook_service` function. Each function can declare an array of arguments that each define the `#name`, `#type`, `#description`, and `#optional` (if the argument is optional) for each argument declared. We can then declare the return value by simply specifying the type that the web service can expect from the return for each web service call.

```
/**
 * Implementation of hook_service()
 */
function voting_service_service() {
  return array(
    // vote.get
    array(
      '#method'   => 'vote.getVote',
      '#callback' => 'voting_service_get_vote',
      '#args'     => array(
        array(
          '#name'         => 'content_type',
          '#type'         => 'string',
          '#description'  => t('The type of content which you are
                                voting for.')),
        array(
          '#name'         => 'content_id',
          '#type'         => 'int',
          '#description'  => t('The ID of the content which you are
                                voting for.')),
        array(
          '#name'         => 'tag',
          '#type'         => 'string',
          '#description'  => t('The category of the vote within the
                                content type.'),
          '#optional'     => TRUE)
      ),
      '#return'   => 'array',
      '#help'     => t('Returns a vote.')
   ),
    // vote.getUserVote
    array(
      '#method'   => 'vote.getUserVote',
      '#callback' => 'voting_service_get_user_vote',
      '#args'     => array(
        array(
          '#name'         => 'content_type',
          '#type'         => 'string',
          '#description'  => t('The type of content which you are
                                voting for.')),
        array(
          '#name'         => 'content_id',
          '#type'         => 'int',
          '#description'  => t('The ID of the content which you are
                                voting for.')),
        array(
          '#name'         => 'tag',
          '#type'         => 'string',
          '#description'  => t('The category of the vote within the
                                content type.'),
          '#optional'     => TRUE)
      ),
      '#return'   => 'array',
      '#help'     => t('Returns a vote.')
   ),
    // vote.set
    array(
      '#method'   => 'vote.setVote',
      '#callback' => 'voting_service_set_vote',
      '#args'     => array(
        array(
          '#name'         => 'content_type',
          '#type'         => 'string',
          '#description'  => t('The type of content which you are
                                voting for.')),
        array(
          '#name'         => 'content_id',
          '#type'         => 'int',
          '#description'  => t('The ID of the content which you are
                                voting for.')),
        array(
          '#name'         => 'vote_value',
          '#type'         => 'int',
          '#description'  => t('The value of the vote.')),
        array(
          '#name'         => 'tag',
          '#type'         => 'string',
          '#description'  => t('The category of the vote within the
                                content type.'),
          '#optional'     => TRUE)
      ),
      '#return'   => 'array',
      '#help'     => t('Submit a new vote.')
   ),
    // vote.delete
    array(
      '#method'   => 'vote.deleteVote',
      '#callback' => 'voting_service_delete_vote',
      '#args'     => array(
        array(
          '#name'         => 'content_type',
          '#type'         => 'string',
          '#description'  => t('The type of content which you are
                                deleting.')),
        array(
          '#name'         => 'content_id',
          '#type'         => 'int',
          '#description'  => t('The ID of the content which you are
                                deleting.')),
        array(
          '#name'         => 'tag',
          '#type'         => 'string',
          '#description'  => t('The category of the vote within the
                                content type.'),
          '#optional'     => TRUE)
      ),
      '#return'   => 'array',
      '#help'     => t('Delete a vote.')
   ),
  );
}
```

Our last and final task is to specify which web service routines should use the API key and which ones should not. By default, each and every method defined, uses the API key. However, for the get vote functions, it is not necessary to require an API key since data is retrieved and not modified when the user gets a vote. Because of this, we can tell the Services module to not require an API key by adding a `#key` parameter for each service and then setting that value to `FALSE`.

```
'#method'   => 'vote.getVote',
'#callback' => 'voting_service_get_vote',
'#key'      => FALSE,

'#method'   => 'vote.getUserVote',
'#callback' => 'voting_service_get_user_vote',
'#key'      => FALSE,
```

At this point, we should have a fully functional custom Voting Service. We can test it out by simply going to the **Services Administrator**, where we should see the following web services defined:



vote

- vote.getVote
- vote.getUserVote
- vote.setVote
- vote.deleteVote

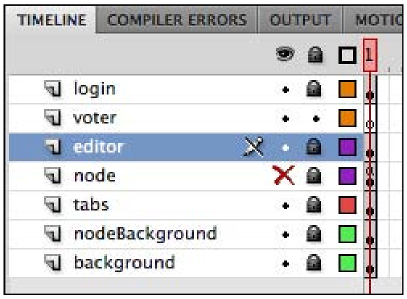
We are now ready to move onto the fun part, *Building a five-star voter in Flash*.

Building a five-star voter in Flash

Before we begin this section, we need to copy the `chapter11` directory and paste that directory and all its contents as the `chapter12` directory, and then rename the `chapter11.fla` to `chapter12.fla`. After we have all the files that we need within our new directory, we can now open up our `chapter12.fla` project file, where we will add a node voter to our editor.

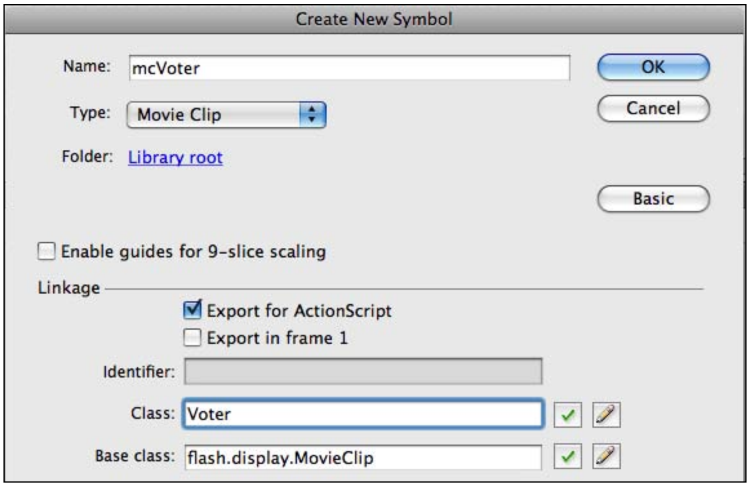
Since our five-star voter is going to be its own separate component within our Flash application, we will start out by creating the movie clip that will serve as the voter, and then build everything from within that movie clip. This will allow us to only see the contents within that movie clip so that we can concentrate fully on the voter and not the node editor in which it will eventually reside.

We will start this out by creating a new layer called **voter** within our node editor timeline, above the **editor** layer.



With the **voter** layer selected, we can now insert a new movie clip by clicking on the menu item **Insert | New Symbol**.

Within the new symbol window, we will give our voter a movie clip name of **mcVoter**, and since we will eventually create an ActionScript class called **Voter** to tie with this movie clip, we can save ourselves a trip by specifying this during the movie clip creation.


Create New Symbol

Name: mcVoter

Type: Movie Clip

Folder: Library root

☐ Enable guides for 9-slice scaling

Linkage

☑ Export for ActionScript

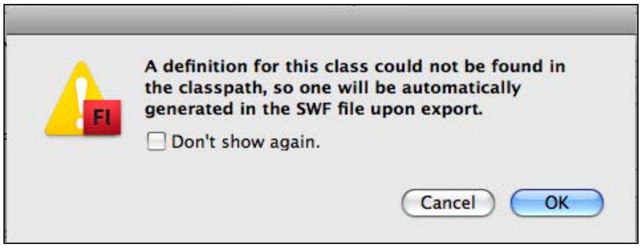
☐ Export in frame 1

Identifier:

Class: Voter

Base class: flash.display.MovieClip

Once we click on **OK**, we should see the following error. This is simply complaining that we have not yet created the Voter class, and basically says that it will create a default one until we specifically define our new class. Since we will eventually build a **Voter** class, we can just ignore this error by clicking on **OK**.



A definition for this class could not be found in the classpath, so one will be automatically generated in the SWF file upon export.

□ Don't show again.

After we click on **OK**, it should create a new symbol, and then present us with a blank stage that we can use to build our five-star voter.

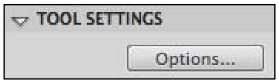
Voter design

Web voters come in all shapes and sizes, and the beauty about using Flash to build ours is that we can literally draw what we want it to look like, and it will automatically function as a voting mechanism. We could easily turn our five-star voter into a five trash can voter by simply drawing trash cans instead of stars. But for the simplicity of this chapter, we will stick with stars since they are familiar.

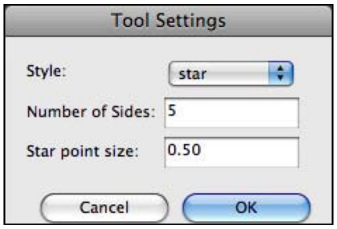
Making some stars

Our first task to build our voter will be to create the stars that we will use for voting. We can do this by first selecting the small black arrow on the Rectangle (▭) tool selection. This should bring up a submenu where we can then select the Polystar tool (⬠).

With this tool selected, we can now create a star by opening up the **Properties** window, and then clicking on the **Options** button inside the **TOOL SETTINGS** section.



This should bring up a new window, where we can tell our polygon to draw a **star** shape with **5** different sides.



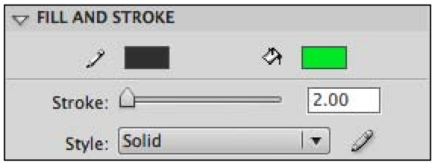
Tool Settings

Style: star

Number of Sides: 5

Star point size: 0.50

Our next task will be to set up the stroke and fill for our stars, which is also found within the **Properties** panel. Since we want to see the outline of our stars clearly when they have not been filled, we will pick a fairly dark color (#333333) and then make the stroke width 2 pixels wide. The fill color, at this moment, can be anything we like since we will later use this fill shape as a mask. Our settings for the fill and stroke should look like the following:



FILL AND STROKE

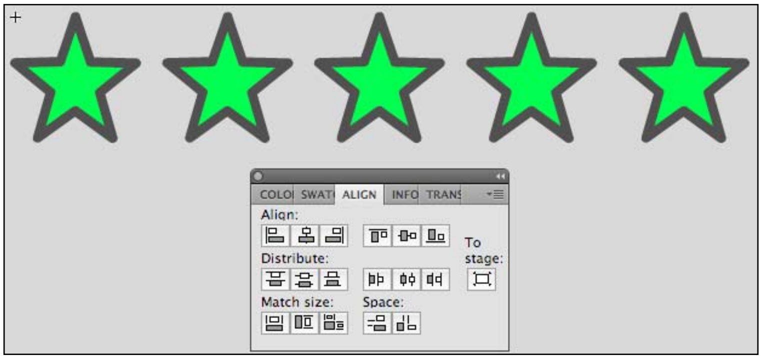
Stroke: 2.00

Style: Solid

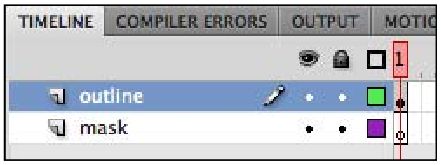
We are now ready to draw our star. With the polygon tool still selected, we will create our star so that it is approximately 25 pixels wide by 25 pixels high. When we are done, it should look similar to the following:

Our next task is to duplicate this shape five times so that we can have a five-star voter. We can do this by selecting the whole star shape, holding down the *Ctrl* key (*Option* for Mac), and then dragging the mouse out to duplicate that shape. We will do this four times, until we have the following:

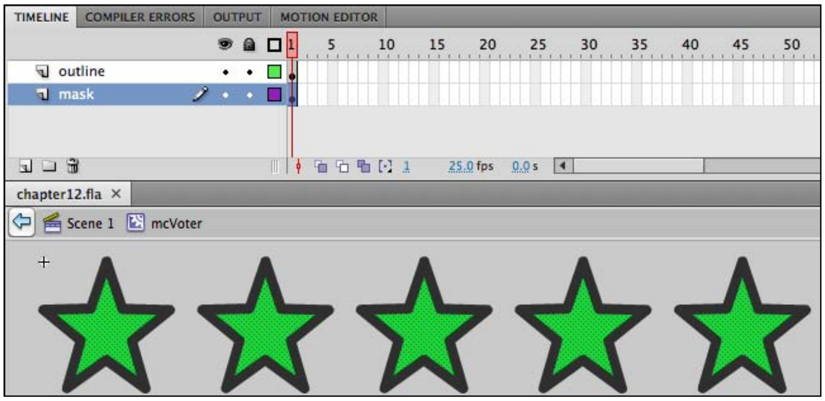
We can make sure that they are evenly spaced by selecting all of the star shapes, and then opening up the **ALIGN** window (menu item **Window | Align**). With the **ALIGN** window open, we will first need to make sure the **To Stage** button is unchecked, and then click on the button that says **Distribute Horizontal Center**. What you should end up with is five evenly spaced stars for voting.



Our next task will be to separate the strokes from the fill regions into two separate layers. We can do this by first creating a new layer called **mask** within the timeline, and then placing that layer above the default layer and renaming it as **outline**.



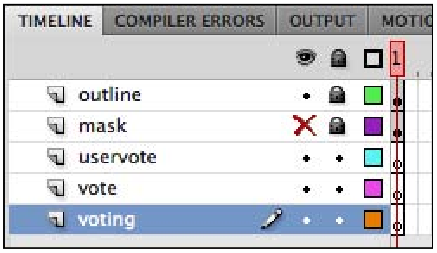
Once we have our layers set up, we can individually click on each of the fill regions for each star (make sure you do not click on the outlines), and then right-click on the mouse and select **Cut**. We will click on the **mask** layer in the **TIMELINE**, then click on the stage, and then right-click and select **Paste in place**. When we are done, we should see the following:



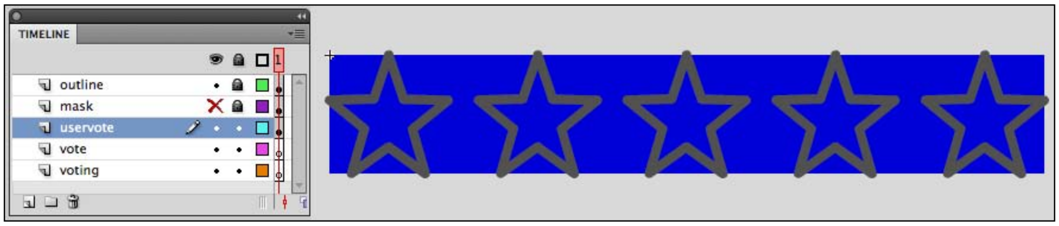
Now that we have our mask layer set up, we will construct our different fill regions for each color that our votes can be.

Adding different vote types

The fill colors for our stars should be dependent on what type of vote we are displaying. For example, if the user has already voted on the piece of content, we will probably want to show the color as blue (this is completely subjective, of course). To show the average vote, we will use yellow. For the votes that light up as the user scrolls their mouse over them, we will use green. Because of this, we will need three different layers below the **mask** layer that will indicate which type of vote we are performing. We will call these different vote types (and layers) **uservote**, **vote**, and **voting**. We will then lock and hide the **mask** layer so that we can work on each one of them individually.

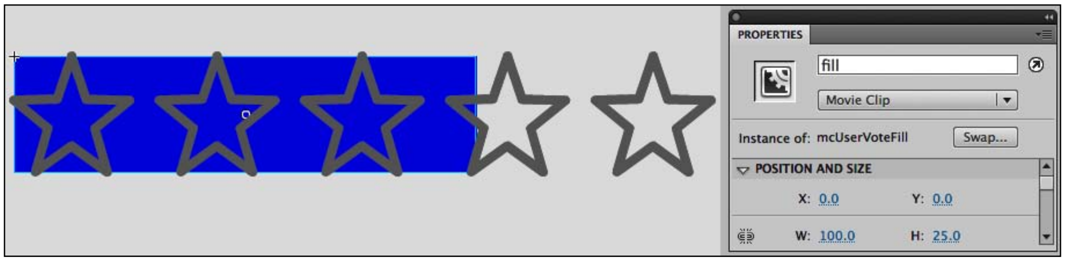


Now, with the **uservote** layer selected, we will simply create a filled rectangle (with no stroke) that will be used to show the user's vote if they have already voted on the piece of content. We need this layer to be a Blue (#0000CC) rectangle that is approximately the height of the stars and the width of all of them combined, like in the following screenshot:



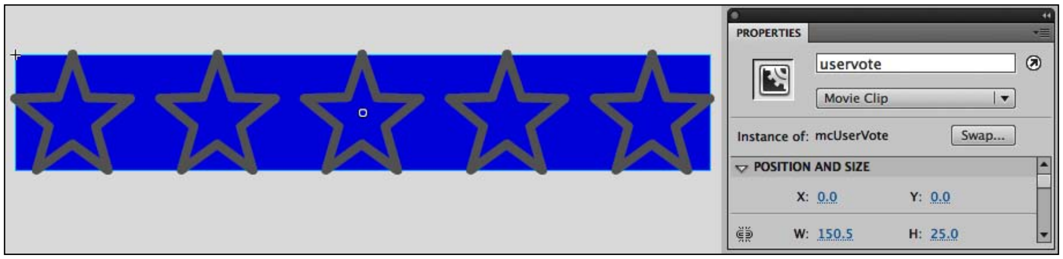
Our next task is to make sure that the width of this rectangle is completely filled when the vote is 100 (the max). This will require a nifty little trick with the movie clip hierarchy, where we will create a child movie clip container for this rectangle that is only 100 pixels wide, and then adjust the parent movie clip to be the size of the star region. This is a clever trick that will do the pixels to vote conversion for us without having to touch any code.

We can start this by first selecting the blue fill region, converting that into a new symbol called **mcUserVoteFill**, and then giving it an instance name of **fill**. We will then give it a width of exactly 100 pixels wide as shown in the following screenshot:



fill
Movie Clip
Instance of: mcUserVoteFill
X: 0.0 Y: 0.0
W: 100.0 H: 25.0

Our next task is to create a new movie clip by first clicking on the **mcUserVoteFill** movie clip and then clicking on the **Modify | Convert to symbol** menu item. We will call this new movie clip **mcUserVote** (basically, a parent of the **mcUserVoteFill** movie clip). We will give this movie clip an instance name of **uservote**, and then stretch it to be the correct width to fill up all of the stars.



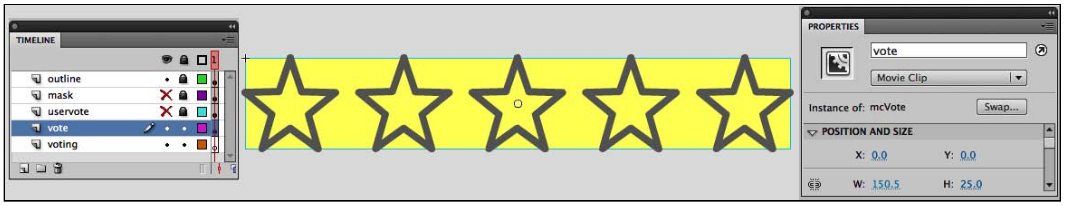
uservote
Movie Clip
Instance of: mcUserVote
X: 0.0 Y: 0.0
W: 150.5 H: 25.0

Now, to change the vote value of this user's vote within ActionScript, all we have to do is change the width of the fill movie clip to be of the same value as the vote, and it will automatically set the width of the parent **uservote** movie clip to show the amount of stars that it equates to! This is a very cool trick that will save us a lot of code and time.

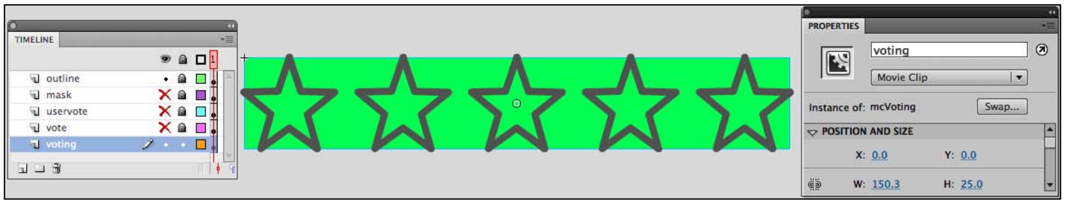
We will do the exact same thing that we did for the user vote, but now, for the **vote** and **voting** layers. Instead of rehashing these steps, I can simply give you the names and instance names of each of the movie clips for each layer so that we can easily replicate this process but for different movie clip names. Indentation within the MovieClip column represents parent-child relationship.

Layer	Movie Clip	Instance
uservote	mcUserVote	uservote
	mcUserVoteFill	fill
vote	mcVote	vote
	mcVoteFill	fill
voting	mcVoting	voting
	mcVotingFill	fill

When we are done, we should have the following:



vote
Movie Clip
Instance of: mcVote
X: 0.0 Y: 0.0
W: 150.5 H: 25.0

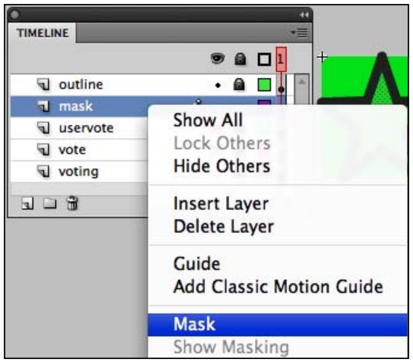


voting
Movie Clip
Instance of: mcVoting
X: 0.0 Y: 0.0
W: 150.3 H: 25.0

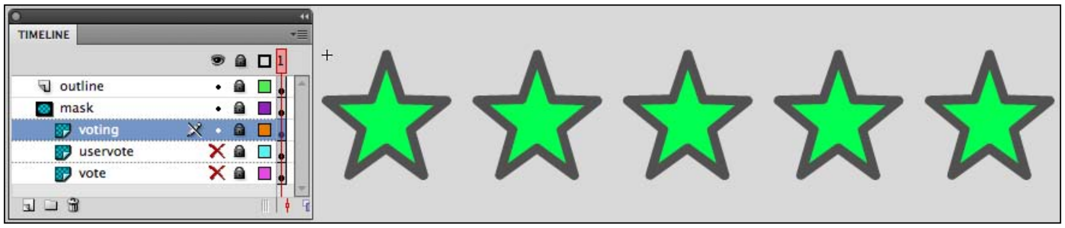
Finally, we can add each of these layers to the mask layer.

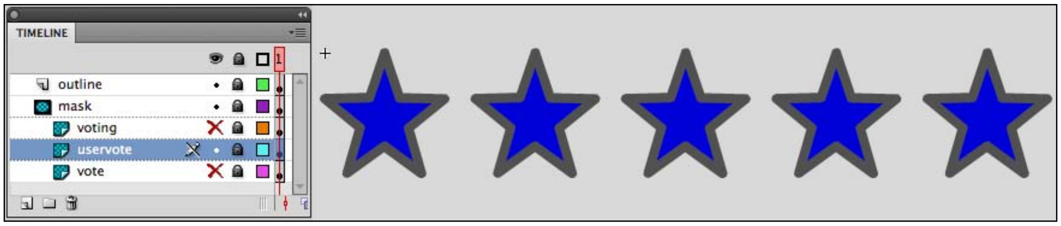
Adding the vote layers to the mask layer

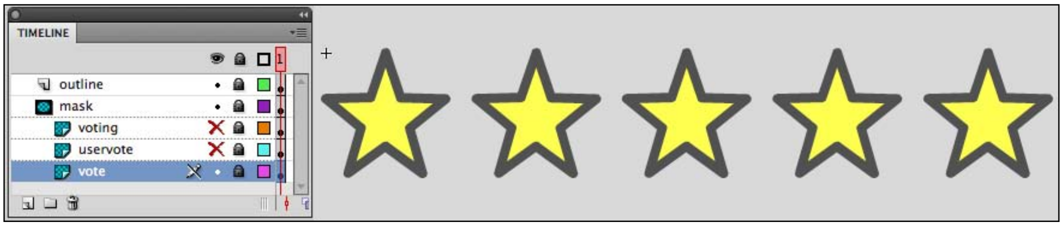
Now that we have all our vote types set up, we can set up the mask layer so that it only shows the colors within the star windows that the mask will define. We can start this by first making the **mask** layer visible, then right-clicking and selecting the **Mask** selection from the menu.



This will add the **uservote** layer to the mask region defined by the mask layer. What we then need to do is move the **vote** and **voting** layers within the mask layer as follows. It is very important that the **voting** layer is on top of all the other voting layers within the mask. Once we lock each of the vote type layers, we should be able to see all of the vote states by clicking on the visible tab for each layer.



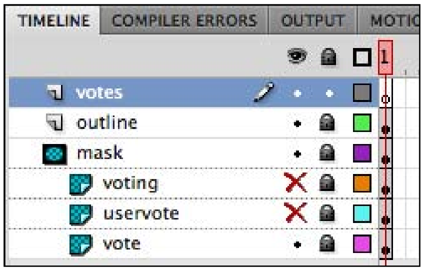




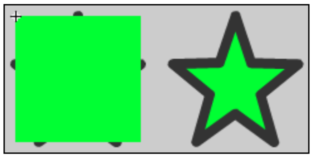
Our next task is to add vote hit regions for when the user wishes to click on a star to cast their vote.

Adding vote hit regions

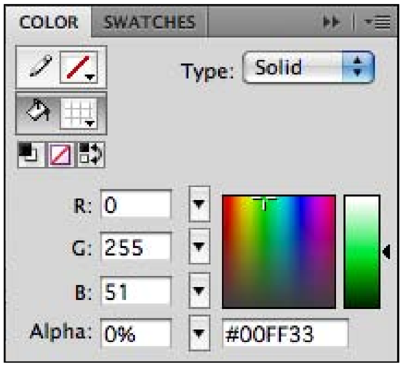
A hit region is simply an invisible button that will define a region that the user can click their mouse on, and it will simulate a button click. This is used primarily as a method for us to cast a vote when the user slides their mouse over the star regions and then presses the star that they wish to vote for. We will start this section out by creating a new layer within our voter called **votes**, and it is important that we make this layer the top most layer in our voter.



Once we have done that, we need to create a 25x25 rectangle with no stroke using the □ tool. It doesn't matter what fill color is used, since we will eventually make the rectangle invisible.



We will now make this rectangle invisible by clicking on the fill region using the selection tool, and then opening up the **Color** window to set the **Alpha** of this color to **0**.



Type: Solid

R: 0

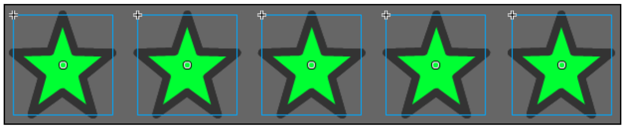
G: 255

B: 51

Alpha: 0%

#00FF33

Our next task is to create a new movie clip from this invisible rectangle by selecting the rectangle and then selecting **Modify | Convert to Symbol** from the Flash menu. We will call this hit region **mcVoteButton**. Our next task is to copy this movie clip so that there is a vote button on top of each star.

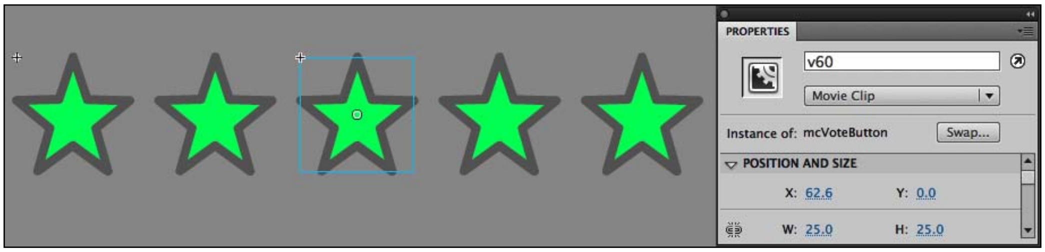


Our last task in this section is to give each of these hit regions an instance name so that we can determine which one is hit when the user clicks on them. Here, we will use a popular technique called name conventions to reduce the amount of code needed to build our voting system.

Using name conventions

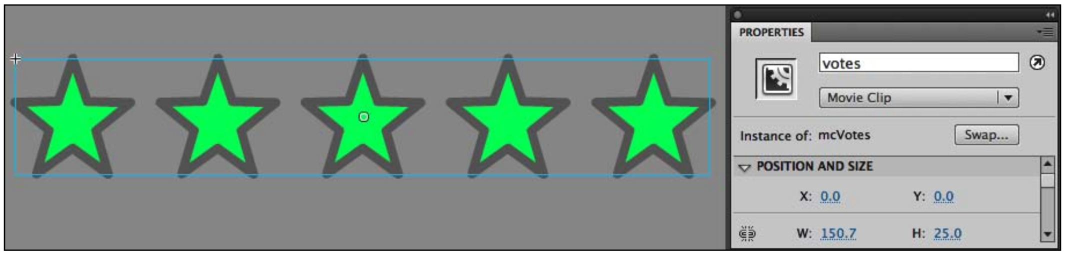
Name conventions is just a fancy term for creating a consistent naming method that can be taken advantage of within the software to simplify the logic behind making decisions on those objects. For example, we could easily name each of our star hit regions, **vote1**, **vote2**, **vote3**, **vote4**, and so on. But then we would need to place some special code within our ActionScript that would switch on each one of these names and then do something different for each one. A better design would be to name our instances so that our ActionScript can use that name to perform its task. Because our ActionScript code is going to translate the hit region object into a vote value for each hit region, why don't we just put the value of the vote within the instance name and then write a single piece of logic to parse out the name and set the value of the vote based on what that name is? Since the first star would equate to the user giving the piece of content a score of 20, why don't we call that object **v20**, and so on? You can probably see where I am going here.

This technique can be very powerful when applied to other software designs, but will be perfect for our need to determine which vote button was pressed and then assign that vote to the content it represents. So, let's now give instance names to our hit regions that represent the value of the vote they represent. These would be **v20**, **v40**, **v60**, **v80**, and **v100** respectively.



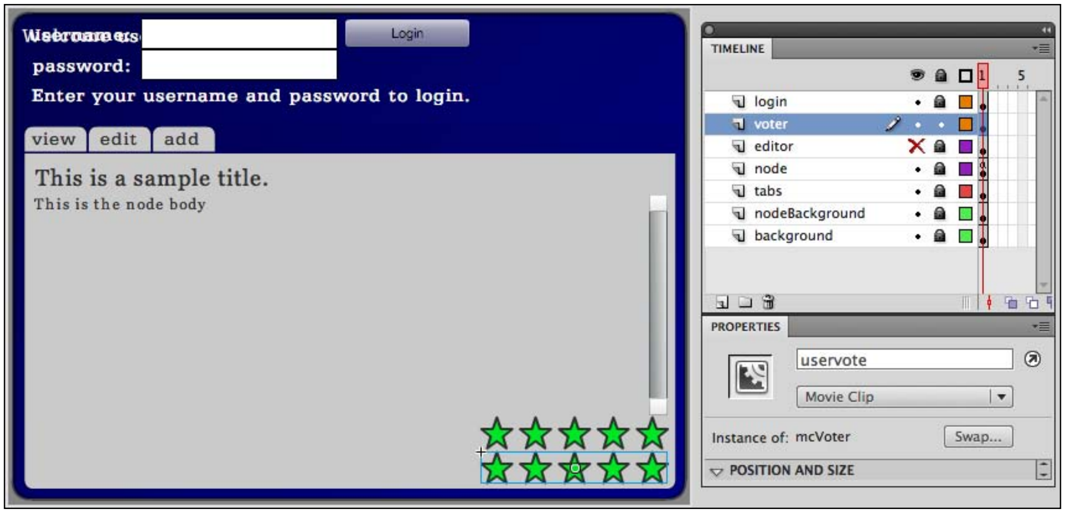
PROPERTIES
v60
Movie Clip
Instance of: mcVoteButton
Swap...
POSITION AND SIZE
X: 62.6 Y: 0.0
W: 25.0 H: 25.0

Once we have given all the votes instance names, our next task is to click on all of the hit regions and then convert them into a single movie clip called **mcVotes** with an instance name of **votes**.



votes
Movie Clip
Instance of: mcVotes
X: 0.0 Y: 0.0
W: 150.7 H: 25.0

After this, we are ready to move back to the root of our stage, where we will place our new five-star voter within the node editor. Since we will need one voter to show the current vote, and then one voter to show the user's vote, we will place two **mcVoter** movie clips right on top of one another, with one called **rating**, and the other called **uservote**. We will then design our Voter class so that it can have two different modes of operation, one for showing the current node vote, and the other to behave as a user voter.



Username:
password:
Enter your username and password to login.
This is a sample title.
This is the node body

login
voter
editor
node
tabs
nodeBackground
background

uservote
Movie Clip
Instance of: mcVoter

If we were to run this project now, we would see that not much happens with the voter. We can change all of that by building the logic behind our Voter movie clip.

Creating a Voter class in ActionScript

Our next task is to build the ActionScript class that will drive the functionality of the voter that we just created within Flash. We can do this by simply creating a new ActionScript file within our `chapter12` directory called `Voter.as` and then starting it off by declaring the class as follows:

```
package
{
   // Import all dependencies
   import flash.display.MovieClip;

   // Declare our Voter class.
   public class Voter extends MovieClip
   {
      // The Voter constructor.
      public function Voter()
      {
         super();
      }

      // Declare all of our child movie clips
      public var votes:MovieClip;
      public var vote:MovieClip;
      public var uservote:MovieClip;
      public var voting:MovieClip;
   }
}
```

We can now move on to initializing our voter.

Initializing the voter

The first thing that we will do to initialize our voter is add an operation mode for the voter. As discussed earlier, our voter can either be a user voter, which will allow each user to see or cast his or her own votes, or it will be a simple vote display that displays the current vote for the piece of content it is connected to. We can do this by setting the default of our player to be in the normal vote mode, and then providing a way to explicitly tell our voter to go into "user mode". This will require a new function called `setUserMode`, which will hide or show certain movie clips within the voter, depending on which mode we are in.

If we are in user mode, then we will want to show the **votes**, **uservote**, and **voting** movie clips, and hide all others. If we are not in user mode, then we will want to show the **vote** movie clip, and hide all others. This can be realized as follows:

```
package
{
   // Import all dependencies
   import flash.display.MovieClip;
   // Declare our Voter class.
   public class Voter extends MovieClip
   {
      // The Voter constructor.
      public function Voter()
      {
         super();
         setUserMode( false );
      }
      // Sets the user mode for this voter.
      public function setUserMode( _user:Boolean )
      {
         // Store the state.
         userMode = _user;
         // Hide or show depending on the mode.
         votes.visible = userMode;
         vote.visible = !userMode;
         uservote.visible = userMode;
         voting.visible = userMode;
      }
      // Declare all of our child movie clips
      public var votes:MovieClip;
      public var vote:MovieClip;
      public var uservote:MovieClip;
      public var voting:MovieClip;
      // Keep track of what mode we are in.
      private var userMode:Boolean;
   }
}
```

We will now want to initialize the fill regions for each of the voter movie clips (**uservote**, **vote**, and **voting**) so that they are zero width. This will basically initialize them to not show any star value when the Voter is initialized.

```
// The Voter constructor.
public function Voter()
{
   super();
   // Initialize our fill widths.
   voting.fill.width = 0;
   uservote.fill.width = 0;
   vote.fill.width = 0;
   // Set the user mode to false.
   setUserMode( true );
}
```

Adding the event handlers

Now that we have our initial state, we need to set up our event handlers to get called when the user interacts with our voter. When the voter is not in user mode, there is not much interaction required since it will simply show the current vote value. However, when the voter is in user mode, we will need to dynamically change the voting movie clip so that it highlights which star the user is currently hovering over. But before we get ahead of ourselves, let's just create the event handlers for each hit region to trigger when the user hovers over or clicks any of those movie clips. We will also want to add a handler when the user exits the voter entirely. For this, we will just trigger the event on the voter rather than each individual hit region.

```
// Import all dependencies
import flash.display.MovieClip;
import flash.events.MouseEvent;
...
...
// Sets the user mode for this voter.
public function setUserMode( _user:Boolean )
{
   // Store the state.
   userMode = _user;
   // Hide or show depending on the mode.
   votes.visible = userMode;
   vote.visible = !userMode;
   uservote.visible = userMode;
   voting.visible = userMode;
```

```
      if( userMode ) {
         // Iterate through all the hit regions.
         for each( var v:MovieClip in votes ) {

            // Setup each hit region for voting.
            v.buttonMode = true;
            v.mouseChildren = false;
            v.addEventListener( MouseEvent.CLICK, onVote );
            v.addEventListener( MouseEvent.MOUSE_OVER, onVoteOver );
         }
         // Called when the mouse exits the voter.
         this.addEventListener( MouseEvent.MOUSE_OUT, onOut );
      }
   }
   // Called when the user makes a vote.
   private function onVote( event:MouseEvent )
   {
      trace( "onVote" );
   }
   // Called when the user hovers over a vote.
   private function onVoteOver( event:MouseEvent )
   {
      trace( "onOver" );
   }
   // Called when the user moves his mouse out.
   private function onOut( event:MouseEvent )
   {
      trace( "onOver" );
   }
```

Handling the voting hover events

Our next task is to move the fill of the voting movie clip so that it matches which star the user is currently hovering over. This is where our naming conventions come in handy. Since we labeled each of the hit regions, the vote value for each movie clip, and then also designed our fill regions so that they can match the value of the vote value, we can very easily assign the fill width to the value of the vote provided from the name of the hit region. All of this can be done by a single line of code within the onVoteOver routine.

```
   // Called when the user hovers over a vote.
   private function onVoteOver( event:MouseEvent )
   {
      voting.fill.width = event.target.name.substr(1);
   }
```

We can then handle the case when the user moves his mouse off the voter by setting the voting fill width to zero when this occurs. This can be done within the `onOut` function as follows:

```
// Called when the user moves his mouse out.
private function onOut( event:MouseEvent )
{
   voting.fill.width = 0;
}
```

At this point, we can temporarily change the `setUserMode( false );` in our constructor to `setUserMode( true );` and then run our application to see a pleasant surprise. The fill width of the voting movie clip changes dynamically depending on which hit region we are over. If we didn't use naming conventions and good design patterns, the amount of code to do this functionality would probably have been much more complicated. After we change the constructor back to `setUserMode( false );` we are now ready to move on!

Getting a vote from Drupal

We are now ready to retrieve the vote information from our Drupal web site and then populate those values in our custom voter. To do this, we will need to add a new function that takes the `DrupalService` component as well as the node ID of the node we are connected to. This function will be called `getVote`, and we will place it right under the `setUserMode` function.

```
// Gets a vote for any node.
public function getVote( _drupal:DrupalService, _nodeId:Number )
{
}
```

We are now ready to use our custom service to retrieve our votes from Drupal. We created two services that we can use within our Flash application to retrieve a vote, one called `vote.getVote` and the other is called `vote.getUserVote`. Since we already have a mode of operation (user mode), we can determine which function to call depending on whether we are in user mode or not. It is also important to note that we need to store the node ID and the `drupal` connection that is passed to this function so that we can reference it again when we make any other Drupal service calls.

As for the callback function, it is declared to take the vote result from our service routine, and then display that vote value in the debug terminal for that instance. This will look as follows:

```
// Gets a vote from Drupal.
public function getVote( _drupal:DrupalService, _nodeId:Number )
{
   // Store the node Id and drupal connection.
   nodeId = _nodeId;
   drupal = _drupal;
   // Get the vote from Drupal.
   var cmd:String = userMode ? "vote.getUserVote" : "vote.getVote";
   drupal.serviceCall( cmd, onGetVote, null, "node", nodeId );
}
// The return function from Drupal.
private function onGetVote( result:Object )
{
   // The value of the result.
   trace( result.value );
}
...
...
// Keep track of what mode we are in.
private var userMode:Boolean;
// Store the node Id.
private var nodeId:Number = 0;
// Store the drupal connection.
private var drupal:DrupalService;
```

Our next task is to change the fill of our `uservote` and `vote` movie clips so that it matches the value of our vote. And since we designed our voter movie clips to perform the vote to pixel conversion for us, this becomes very simple.

```
// The return function from Drupal.
private function onGetVote( result:Object )
{
   // Set the fill width of our voting movie clips to the value.
   if( userMode ) {
      uservote.fill.width = result.value;
   }
   else {
      vote.fill.width = result.value;
   }
}
```

And we are now ready to move onto setting votes in Drupal.

Setting a vote in Drupal

In order for us to use the vote.setVote and vote.deleteVote functions, we will need to make some changes to our DrupalService class since these functions are protected with the API key. This can be done by simply adding the following lines of code to the usesKey function within our DrupalService class, within the **DrupalService.as** file:

```
// Determines if we should use an API Key.
private function usesKey( command:String ):Boolean
{
   switch ( command ) {
      case "user.login":
      case "user.logout":
      case "node.save":
      case "vote.setVote":
      case "vote.deleteVote":
         return true;
         break;
   }
   return false;
}
```

With that out of the way, we are ready to dive into the Voter class, where we will add the ability to set the votes that the user selects. Surprisingly, this is going to be very trivial since most of the hard stuff is handled from the Voting API within our Drupal web site. Because of this, we simply need to handle the case when the user clicks on a hit region, which calls onVote, and then pass the same callback that sets the vote value when the server returns the result. All of this functionality can be provided within the onVote function, which gets triggered when the user clicks on any hit region within our voter.

```
// Called when the user makes a vote.
private function onVote( event:MouseEvent )
{
   // Check to make sure we know what node we are.
   if( nodeId )
   {
      // Get the value of the vote that was clicked.
      var voteValue:String = event.target.name.substr(1);
      drupal.serviceCall( "vote.setVote", onGetVote, null, "node",
      nodeId, voteValue );
   }
}
```

We are now ready to add this functionality to our main.as file.

Adding the voters to main.as

This section is going to be surprisingly simple. Because we encapsulated all of the voting functionality within the Voter class, we simply need to set the user state for the uservoter movie clip, and then get the votes when each node loads in our node editor. This is the beauty of object-oriented design. It makes the development and interfaces between different software components much easier. To start, we will simply set the user mode of the uservote movie clip at the beginning of the main.as file.

```
...
// The current user.
var user:Object = null;
var currentNode:Object = null;
// Set the uservote to user mode.
uservote.setUserMode( true );
...
...
```

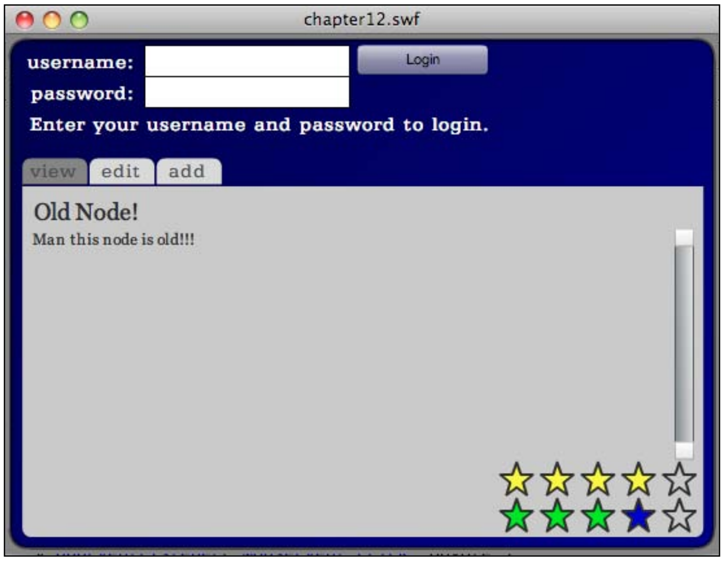
Next, we will make it such that the voting system is only visible on the **view** tab. We can do this within the selectTab routine by placing the following code:

```
// Selects a new tab.
function selectTab( newTab:MovieClip )
{
   // Make the previous tab go to normal.
   selectedTab.gotoAndStop("normal");
   // Set the new selected tab.
   selectedTab = newTab;
   // Go to the selected state.
   selectedTab.gotoAndStop( "selected" );
   // Hide or show the editor or node view.
   nodeMC.visible = (selectedTab.name == "view");
   editor.visible = (selectedTab.name != "view");
   uservote.visible = (selectedTab.name == "view");
   rating.visible = (selectedTab.name == "view");
   if( selectedTab.name == "edit" && currentNode ) {
      editor.title.text = currentNode.title;
      editor.body.htmlText = currentNode.body;
   }
   else {
      editor.title.text = "";
      editor.body.htmlText = "";
   }
}
```

And finally, we can hook up all the functionality by simply calling the getVote routine on both the uservote and rating movie clips once the node is done loading. This would place these calls within the onNodeLoad function as follows:

```
// Called when Drupal returns with our node.
function onNodeLoad( node:Object )
{
   // Store the current node.
   currentNode = node;

   // Get the rating and user votes for this node.
   rating.getVote( drupal, node.nid );
   uservote.getVote( drupal, node.nid );

   // Set the string of our body and title fields.
   nodeMC.title.text = node.title;
   nodeMC.body.htmlText = node.body;

   // Update the node body scroll bar.
   nodeMC.bodyScroll.update();
```

And that's it! We should now be able to run our application and see a fully functional voting system.



Enter your username and password to login.

Old Node!

Man this node is old!!!

Summary

Now that we have built a complete Flash-Drupal application from the ground up, I believe that you have all the knowledge necessary to tackle any Flash and Drupal project thrown your way. The five-star voter was a perfect exercise to take you through the complete development process of building a custom Flash application for Drupal. We started by building a custom service to tap into the power of the Services module to provide server-side functionality to our voting application. After that, we started from scratch in creating the five-star voter in Flash, and then employed object-oriented techniques to hook up that movie clip with business logic to allow remote Drupal voting.

With all the lessons learned in this book, I really do hope that you now have all the tools necessary to create any custom Flash application that will tap into the power and flexibility of Drupal. We have explored many popular uses for this technology from media handling to hybrid Flash-Drupal architectures, and hopefully, this has completely opened the door to many new, unique, and innovative Flash applications. The possibilities are really limitless. I hope that, after reading through these lessons, you now fully understand how powerful the combination of these two amazing technologies can really be, and hope that one day, one of you will use this knowledge to create the next big thing.

Index

Thank you for buying

Flash with Drupal

Packt Open Source Project Royalties

When we sell a book written on an Open Source project, we pay a royalty directly to that project. Therefore by purchasing Flash with Drupal, Packt will have given some of the money received to the Drupal project.

In the long term, we see ourselves and you—customers and readers of our books—as part of the Open Source ecosystem, providing sustainable revenue for the projects we publish on. Our aim at Packt is to establish publishing royalties as an essential part of the service and support a business model that sustains Open Source.

If you're working with an Open Source project that you would like us to publish on, and subsequently pay royalties to, please get in touch with us.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

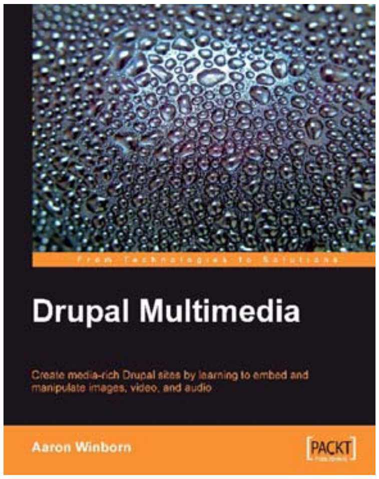
We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.PacktPub.com`.

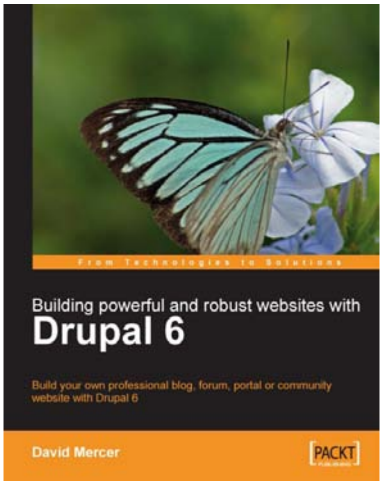

From Technologies to Solutions
Drupal Multimedia
Create media-rich Drupal sites by learning to embed and manipulate images, video, and audio
Aaron Winborn
PACKT

Drupal Multimedia

ISBN: 978-1-847194-60-2 Paperback: 241 pages

Create media-rich Drupal sites by learning to embed and manipulate images, video, and audio

1. Learn to integrate multimedia in your Drupal websites
2. Find your way round contributed modules for adding media to Drupal sites
3. Tackle media problems from all points of views: content editors, administrators, and developers


From Technologies to Solutions
Building powerful and robust websites with
Drupal 6
Build your own professional blog, forum, portal or community website with Drupal 6
David Mercer
PACKT

Building Powerful and Robust Websites with Drupal 6

ISBN: 978-1-847192-97-4 Paperback: 362 pages

Build your own professional blog, forum, portal or community website with Drupal 6

1. Set up, configure, and deploy Drupal 6
2. Harness Drupal's world-class Content Management System
3. Design and implement your website's look and feel
4. Easily add exciting and powerful features
5. Promote, manage, and maintain your live website

Please check **www.PacktPub.com** for information on our titles