



Reference Guide

ColdFusion on Wheels 1.1

Updated March 13, 2011

Table of Contents

Introduction

- Frameworks and Wheels
- Requirements
- Installation
- Upgrading to Wheels 1.1.x
- Beginner Tutorial: Hello World
- Beginner Tutorial: Hello Database
- Tutorial: Wheels, AJAX, and You

Working with Wheels

- Conventions
- Configuration and Defaults
- Directory Structure
- Switching Environments
- Using and Creating Plugins

Handling Requests with Controllers

- Request Handling
- Rendering Content
- Redirecting Users
- Sending Files
- Sending Email
- Responding with Multiple Formats
- Using the Flash
- Filters
- Verification
- Event Handlers
- URL Rewriting
- Using Routes
- Obfuscating URLs

- Caching

Database Interaction Through Models

- Object Relational Mapping
- Creating Records
- Reading Records
- Updating Records
- Deleting Records
- Column Statistics
- Dynamic Finders
- Getting Paginated Data
- Associations
- Nested Properties
- Object Validation
- Object Callbacks
- Calculated Properties
- Transactions
- Dirty Records
- Soft Delete
- Automatic Time Stamps
- Using Multiple Data Sources

Displaying Views to Users

- Pages
- Partials
- Linking Pages
- Using Layouts
- Form Helpers and Showing Errors
- Displaying Links for Pagination
- Date, Media, and Text Helpers
- Creating Your Own View Helpers

Frameworks and Wheels

Learn the goals of ColdFusion on Wheels as well as web development frameworks in general. Then learn about Wheels's goals and some key concepts.

This chapter will introduce you to frameworks in general and later specifically to ColdFusion on Wheels. We'll help you decide if you even need a framework at all and what common problems a framework tries to solve. If we're able to convince you that using a framework is the right thing for you, then we'll present our goals with creating Wheels and show you some key Wheels concepts.

So let's get started.

Do I Really Need to Use a Framework?

Short answer, no. If you don't mind doing the same thing over and over again and are getting paid by the hour to do so, then by all means keep doing that. ;)

Slightly longer answer, no. If you're working on a highly customized project that does not fall within what 9 out of 10 web sites/applications normally do then you likely need a high percentage of custom code, and a framework will not help much.

However, if you're like most of us and have noticed that for every new project you start on—or even every new feature you add to an existing project—you waste a lot of time re-creating the wheel, then you should read on because Wheels may just be the solution for you!

Wheels will make starting a new project or building a new feature quick and painless. You can get straight to solving business problems on day one! To understand how this is achieved, we figured that a little background info on frameworks in general may help you out.

All good frameworks rise from the need to solve real problems in real world situations. Wheels is based heavily on the Rails framework for Ruby and also gets inspiration from Django and, though to a lesser extent, other frameworks in the ColdFusion space (like Fusebox, for example). Over the years the contributors to these frameworks have identified problems and tedious tasks in their own development processes, built a solution for it, and abstracted (made it more generic so it suits any project) the solution into the framework in question. Piggy-backing on what all these great programmers have already created and adding a few nice solutions of our own, Wheels stands on solid ground.

OK, so that was the high level overview of what frameworks are meant to do. But let's get a little more specific.

Framework Goals in General

Most web development frameworks set out to address some or all of these common concerns:

- Map incoming requests to the code that handles them.
- Separate your business logic from your presentation code.
- Let you work at a higher level of abstraction, thus making you work faster.

- Give you a good code organization structure to follow.
- Encourage clean and pragmatic design.
- Simplify saving data to a storage layer.

Like all other good frameworks, Wheels does all this. But there are some subtle differences, and certain things are more important in Wheels than in other frameworks and vice versa. Let's have a look at the specific goals with Wheels so you can see how it relates to the overall goals of frameworks in general.

Our Goals With Wheels

As we've said before, Wheels is heavily based on Ruby on Rails, but it's not a direct port, and there are some things that have been changed to better fit the CFML language. Here's a brief overview of the goals we're striving for with Wheels (most of these will be covered in greater detail in later chapters):

Simplicity

We strive for simplicity on a lot of different levels in Wheels. We'll gladly trade code beauty in the framework's internal code for simplicity for the developers who will use it. This goal to keep things simple is evident in a lot of different areas in Wheels. Here are some of the most notable ones:

- The concept of object oriented programming is very simple and data-centric in Wheels, rather than 100% "pure" at all times.
- By default, you'll always get a query result set back when dealing with multiple records in Wheels, simply because that is the way we're all used to outputting data.
- Wheels encourages best practices, but it will never give you an error if you go against any of them.
- With Wheels, you won't program yourself into a corner. If worse comes to worst, you can always drop right out of the framework and go back to old school code for a while if necessary.
- Good old CFML code is used for everything, so there is no need to mess with XML for example.

What this means is that you don't have to be a fantastic programmer to use the framework (although it doesn't hurt). It's enough if you're an average programmer. After using Wheels for a while, you'll probably find that you've become a better programmer though!

Documentation

If you've ever downloaded a piece of open source software, then you know that most projects lack documentation. Wheels hopes to change that. We're hoping that by putting together complete, up-to-date documentation that this framework will appeal, and be usable, by everyone. Even someone who has little ColdFusion programming background, let alone experience with frameworks.

Key Wheels Concepts

Besides what is already mentioned above, there are some key concepts in Wheels that makes sense to familiarize yourself with early on. If you don't feel that these concepts are to your liking, feel free to look for a different framework or stick to using no

framework at all. Too often programmers choose a framework and spend weeks trying to bend it to do what they want to do rather than follow the framework conventions.

Speaking of conventions, this brings us to the first key concept:

Convention Over Configuration

Instead of having to set up tons of configuration variables, Wheels will just assume you want to do things a certain way by using default settings. In fact, you can start programming a Wheels application without setting any configuration variables at all!

If you find yourself constantly fighting the conventions, then that is a hint that you're not yet ready for Wheels or Wheels is not ready for you. ;)

Beautiful Code

Beautiful (for lack of a better word) code is code that you can scan through and immediately see what it's meant to do. It's code that is never repeated anywhere else. And, most of all, it's code that you'll enjoy writing and will enjoy coming back to 6 months from now.

Sometimes the Wheels structure itself encourages beautiful code (separating business logic from request handling, for example). Sometimes it's just something that comes naturally after reading documentation, viewing other Wheels applications, and talking to other Wheels developers.

Model-View-Controller (MVC)

If you've investigated frameworks in the past, then you've probably heard this terminology before. Model-View-Controller, or MVC, is a way to structure your code so that it is broken down into three easy-to-manage pieces:

- **Model:** Just another name for the representation of data, usually a database table.
- **View:** What the user or their browser sees and interacts with (a web page in most cases).
- **Controller:** The behind-the-scenes guy that's coordinating everything.

"Uh, yeah. So what's this got to do with anything?" you may ask. MVC is how Wheels structures your code for you. As you start working with Wheels applications, you'll see that most of the code you write (database queries, forms, and data manipulation) are very nicely separated into one of these three categories.

The benefits of MVC are limitless, but one of the major ones is that you almost always know right where to go when something needs to change.

If you've added a column to the vehicles table in your database and need to give the user the ability to edit that field, all you need to change is your View. That's where the form is presented to the user for editing.

If you find yourself constantly getting a list of all the red cars in your inventory, you can add a new method to your Model called `getRedCars()` that does all the work for you. Then when you want that list, just add a call to that method in your Controller and you've got 'em!

Object Relational Mapping (ORM)

The Object Relational Mapping, or ORM, in Wheels is perhaps the one thing that could potentially speed up your development the most. An ORM handles mapping objects in memory to how they are stored in the database. It can replace a lot of your query writing with simple methods such as `user.save()`, `blogPost.comments(order="date")`, and so on. We'll talk a lot more about the ORM in Wheels in the chapter on models.

There's Your Explanation

So there you have it, a completely fair and unbiased introduction to Wheels. ;)

If you've been developing ColdFusion applications for a while, then we know this all seems hard to believe. But trust us; it works. And if you're new to ColdFusion or even web development in general, then you probably aren't aware of most of the pains that Wheels was meant to alleviate!

That's okay. You're welcome in the Wheels camp just the same.

Requirements

What you need to know and have installed before you start programming in Wheels.

We can identify 3 different types of requirements that you should be aware of:

1. **Project Requirements.** Is Wheels a good fit for your project?
2. **Developer Requirements.** Do you have the knowledge and mindset to program effectively in Wheels?
3. **System Requirements.** Is your server ready for Wheels?

1. Project Requirements

Before you start learning Wheels and making sure all the necessary software is installed on your computer you really need to take a moment and think about the project you intend to use Wheels on. Is it a ten page website that won't be updated very often? Is it a space flight simulator program for NASA? Is it something in between?

Most websites are, at their cores, simple data manipulation applications. You fetch a row, make some updates to it, stick it back in the database and so on. This is the "target market" for Wheels - simple CRUD (create, read, update, delete) website applications.

A simple ten page website won't do much data manipulation, so you don't need Wheels for that (or even ColdFusion in some cases). A flight simulator program will do so much more than simple CRUD work, so in that case, Wheels is a poor match for you (and so is ColdFusion).

If your website falls somewhere in between these two extreme examples, then read on. If not, go look for another programming language and framework. ;)

Another thing worth noting right off the bat (and one that ties in with the simple CRUD reasoning above) is that Wheels takes a very data-centric approach to the development process. What we mean by that is that it should be possible to visualize and implement the database design early on in the project's life cycle. So, if you're about to embark on a project with an extensive period of object oriented analysis and design which, as a last step almost, looks at how to persist objects, then you should probably also look for another framework.

Still reading?

Good. :)

Moving on...

2. Developer Requirements

Yes, there are actually some things you should familiarize yourself with before starting to use Wheels. Don't worry though. You don't need to be an expert on any one of them. A basic understanding is good enough.

- **CFML.** You should know CFML, the ColdFusion programming language. (Surprise!)

- **Object Oriented Programming.** You should grasp the concept of object oriented programming and how it applies to ColdFusion.
- **Model-View-Controller.** You should know the theory behind the Model-View-Controller development pattern.

CFML

Simply the best web development language in the world! The best way to learn it, in our humble opinion, is to get the free developer edition of Adobe ColdFusion, buy Ben Forta's ColdFusion Web Application Construction Kit series, and start coding using your programming editor of choice.

Object Oriented Programming (OOP)

This is a programming methodology that uses constructs called *objects* to design applications. Objects model real world entities in your application. OOP is based on several techniques including *inheritance*, *modularity*, *polymorphism*, and *encapsulation*. Most of these techniques are supported in CFML, making it a fairly functional object oriented language. At the most basic level, a `.cfc` file in ColdFusion is a class, and you create an instance of a class by using the `CreateObject` function or the `<cfobject>` tag.

Trying to squeeze an explanation of object oriented programming and how it's used in CFML into a few sentences is impossible, and a detailed overview of it is outside the scope of this chapter. There is lots of high quality information online, so go ahead and Google it.

Model-View-Controller

Model-View-Controller, or MVC for short, is a way to structure your code so that it is broken down into 3 easy-to-manage pieces:

- **Model.** Just another name for the representation of data, usually a database table.
- **View.** What the user sees and interacts with (a web page in our case).
- **Controller.** The behind-the-scenes guy that's coordinating everything.

MVC is how Wheels structures your code for you. As you start working with Wheels applications, you'll see that most of the code you write is very nicely separated into one of these 3 categories.

3. System Requirements

Wheels requires that you use one of these CFML engines:

- Adobe ColdFusion 8.0.1
- Adobe ColdFusion 9
- Railo (version 3.1.2.020 or greater)

Wheels makes heavy use of CFML's `OnMissingMethod` event, which wasn't available until the release of CF 8.

Operating Systems

Your setup with ColdFusion 8 and Wheels can then be installed on **Windows, Mac OS X, UNIX, or Linux**—they all work just fine.

Web Servers

You also need a web server. Wheels runs on all popular web servers, including **Apache**, Microsoft **IIS**, **Jetty**, and the **JRun** web server that ships with Adobe ColdFusion. Some web servers support URL rewriting out of the box, some support the `cgi.path_info` variable which is used to achieve partial rewriting, and some don't have support for either.

Don't worry though. Wheels will adopt to your setup and run just fine, but the URLs that it creates might differ a bit. You can read more about this in the URL Rewriting chapter.

Database Engines

Finally, to build any kind of meaningful website application, you will likely interact with a database. Currently supported databases are SQL Server 7 or later, Oracle 10g or later, MySQL 5, PostgreSQL, and H2.

Regarding MySQL, please note that MySQL 4 is not supported. We also recommend using the InnoDB engine if you want for Transactions to work.

OK, hopefully this chapter didn't scare you too much. You can move on knowing that you have the basic knowledge needed, the software to run Wheels on, and a suitable project to start with.

Installation

Instructions for installing Wheels on your system.

Installing Wheels is so simple that there is barely a need for a chapter devoted to it. But we figured we'd better make one anyway in case anyone is specifically looking for a chapter about installation.

So, here are the simple steps you need to follow to get rolling on Wheels...

1. Download Wheels

You have 2 choices when downloading Wheels. You can either use the latest official release of Wheels, or you can take a walk on the wild side and go with the latest committed source code in our Subversion (SVN) repository.

The latest official release can be found in the downloads section of this website, and the Git repository is available at our GitHub repo.

In most cases, we recommend going with the official release because it's well documented and has been through a lot of bug testing. Only if you're in desperate need of a feature that has not been released yet would we advise you to go with the version stored in the SVN trunk.

Let's assume you have downloaded the latest official release. (Really, you should go with this option.) You now have a `.zip` file saved somewhere on your computer. On to the next step...

2. Setup the Website

Getting an empty website running with Wheels installed is an easy process if you already know your way around IIS or Apache. Basically, you need to create a new website in your web server of choice and unzip the contents of the file into the root of it.

In case your not sure, here are the instructions for setting up an empty Wheels site that can be accessed when typing `localhost` in your browser. The instructions refer to a system running Windows Server 2003 and IIS, but you should be able to follow along and apply the instructions with minor modifications to your system. (See Requirements for a list of tested systems).

- Create a new folder under your web root (usually `C:\Inetpub\wwwroot`) named `wheels_site` and unzip the Wheels `.zip` file into the root of it.
- Create a new website using IIS called `Wheels Site` with `localhost` as the host header name and `C:\Inetpub\wwwroot\mysite` as the path to your home directory.

If you want to run a Wheels powered application from a subfolder in an existing website, this is entirely possible, but you may need to get a little creative with your URL rewrite rules if you want to get pretty URLs—it will only work out of the box on recent versions of Apache. (Read more about this in the URL Rewriting chapter.)

3. Setup the Database (Optional)

Create a new database in MySQL, Oracle, PostgreSQL, Microsoft SQL Server, or H2 and add a new data source for it in the ColdFusion/Railo Administrator, just as you'd normally do. Now open up `config/settings.cfm` and call `set(dataSourceName=" ")` with the name you chose for the data source.

If you don't want to be bothered by opening up a Wheels configuration file at all, there is a nice convention you can follow for the naming. Just name your data source with the same name as the folder you are running your website from (`mysite` in the example above), and Wheels will use that when you haven't set the `dataSourceName` variable using the `set()` function.

4. Test It

When you've followed the steps above, you can test your installation by typing `http://localhost/` (or whatever you set as the host header name) in your web browser. You should get a page saying "Welcome to Wheels!"

That's it. You're done. This is where the fun begins!

Upgrading to Wheels 1.1.x

Instructions for upgrading your ColdFusion on Wheels 1.0.x application to Wheels 1.1.x.

If you are upgrading from Wheels 1.0 or newer, the easiest way to upgrade is to replace the `wheels` folder with the new one from the 1.1 download. If you are upgrading from an earlier version, we recommend reviewing the steps outlined in Upgrading to Wheels 1.0.

Note: To accompany the newest 1.1.x releases, we've highlighted the changes that are affected by each release in this cycle.

Plugin Compatibility

Be sure to review your plugins and their compatibility with your newly-updated version of Wheels. Some plugins may stop working, throw errors, or cause unexpected behavior in your application.

Supported System Changes

- 1.1: The minimum Adobe ColdFusion version required is now 8.0.1.
- 1.1: The minimum Railo version required is now 3.1.2.020.
- 1.1: The H2 database engine is now supported.

File System Changes

- 1.1: The `.htaccess` file has been changed. Be sure to copy over the new one from the new version 1.1 download and copy any additional changes that you may have also made to the original version.

Database Structure Changes

- 1.1: By default, Wheels 1.1 will wrap database queries in transactions. This requires that your database engine supports transactions. For MySQL in particular, you can convert your MyISAM tables to InnoDB to be compatible with this new functionality. Otherwise, to turn off automatic transactions, place a call to `set(transactionMode="none")`.
- 1.1: Binary data types are now supported.

CFML Code Changes

Model Code

- **1.1:** Validations will be applied to some model properties automatically. This may cause unintended behavior with your validations. To turn this setting off, call `set(automaticValidations=false)` in `config/settings.cfm`.
- **1.1:** The `class` argument in `hasOne()`, `hasMany()`, and `belongsTo()` has been deprecated. Use the `modelName` argument instead.
- **1.1:** `afterFind()` callbacks no longer require special logic to handle the setting of properties in objects and queries. (The "query way" works for both cases now.) Because `arguments` will always be passed in to the method, you can't rely on `StructIsEmpty()` to determine if you're dealing with an object or not. In the rare cases that you need to know, you can now call `isInstance()` or `isClass()` instead.
- **1.1:** On create, a model will now set the `updatedAt` auto-timestamp to the same value as the `createdAt` timestamp. To override this behavior, call `set(setUpdatedAtOnCreate=false)` in `config/settings.cfm`.

View Code

- **1.1:** Object form helpers (e.g. `textField()` and `radioButton()`) now automatically display a `label` based on the property name. If you left the `label` argument blank while using an earlier version of Wheels, some labels may start appearing automatically, leaving you with unintended results. To stop a label from appearing, use `label=false` instead.
- **1.1:** The `contentForLayout()` helper to be used in your layout files has been deprecated. Use the `includeContent()` helper instead.
- **1.1:** In production mode, query strings will automatically be added to the end of all asset URLs (which includes JavaScript includes, stylesheet links, and images). To turn off this setting, call `set(assetQueryString=false)` in `config/settings.cfm`.
- **1.1:** `stylesheetLinkTag()` and `javascriptIncludeTag()` now accept external URLs for the `source/ sources` argument. If you manually typed out these tags in previous releases, you can now use these helpers instead.
- **1.1:** `flashMessages()`, `errorMessageOn()`, and `errorMessagesFor()` now create camelCased `class` attributes instead (for example `error-messages` is now `errorMessages`). The same goes for the `class` attribute on the tag that wraps form elements with errors: it is now `fieldWithErrors`.

Controller Code

- **1.1.1:** The `if` argument in all validation functions is now deprecated. Use the `condition` argument instead.

Beginner Tutorial: Hello World

In this tutorial, we'll be writing a simple application to make sure we have Wheels installed properly and that everything is working as it should. Along the way, you'll get to know some basics about how applications built on top of Wheels work.

Testing Your Install

Let's make sure we're all on the same page. I'm going to assume that you've already downloaded the latest version of Wheels and have it installed on your system. If you haven't done that, stop and read the Installation chapter and get everything setup. It's okay, this web page will wait for you.

Okay, so you have Wheels installed and can see the Wheels "Congratulations!" page as shown in *Figure 1* below. That wasn't that hard now, was it?

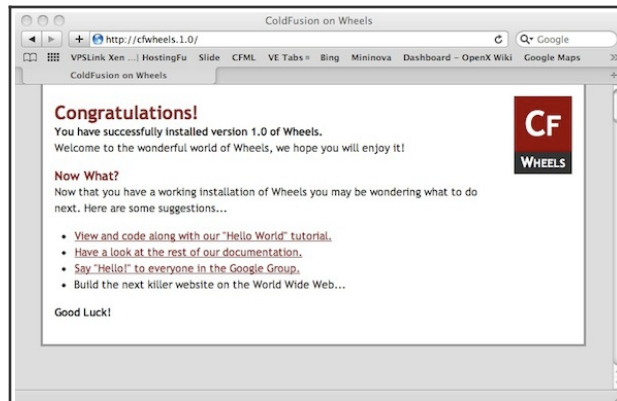


Figure 1: Wheels congratulations screen.

Hello World: Your First Wheels App

Okay, let's get to some example code. We know that you've been dying to get your hands on some code!

To continue with Programming Tutorial Tradition, we'll create the ubiquitous *Hello World!* application. But to keep things interesting, let's add a little Wheels magic along the way.

Setting up the Controller

Let's create a controller from scratch to illustrate how easy it is to set up a controller and plug it into the Wheels framework.

First, create a file called `Say.cfc` in the `controllers` directory and add the code below to the file.

```
<cfcomponent extends="Controller">
</cfcomponent>
```

Congratulations, you just created your first Wheels controller! What does this controller do, you might ask? Well, to be honest, not much. It has no functions defined so it doesn't add any new functionality to our Wheels application. But because it extends the base `Controller` component, it inherits quite a bit of powerful functionality and is now tied into our Wheels application.

So what happens if we try to call our new controller right now? Lets take a look! Open your browser and point your browser to the new controller. Because my server is installed on port 80, myURL is `http://cfwheels.1.0/index.cfm/say`. You may need to enter a different URL, depending on how your web server is configured. In my case, I will be using the host name `cfwheels.1.0`, so my URL will look like:

```
http://cfwheels.1.0/index.cfm/say
```

If you did everything right, you will get the Wheels error shown below in *Figure 2*. (Ironic that getting an error is doing something right, huh? Don't get used to it, buddy!)

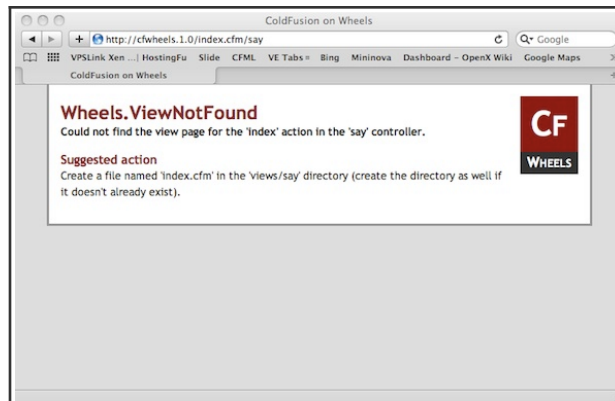


Figure 2: Wheels error after setting up your blank say controller.

The error says "Could not find the view page for the 'index' action in the 'say' controller." Where did "index" come from? The URL we typed in only specified a controller name but no action. When an action is not specified in the URL, Wheels assumes that we want the default action. Out of the box, the default action in Wheels is set to `index`. So in our example, Wheels tried to find the `index` action within the `say` controller, and it threw an error because it couldn't find its view page.

Setting up an Action

But let's jump ahead. Now that we have the controller created, let's add an action to it called `hello`. Change your `say` controller so it looks like the code block below:

```
<cfcomponent extends="Controller">
    <cffunction name="hello"></cffunction>
```

```
</cfcomponent>
```

As you can see, we created an empty method named `hello`.

Now let's call our new action in the browser and see what we get. To call the `hello` action we simply add `/hello` to the end of the previous URL that we used to call our `say` controller:

```
http://cfwheels.1.0/index.cfm/say/hello
```

Once again, we get a ColdFusion error. Although we have created the controller and added the `hello` action to it, we haven't created the `view`.

Setting up the View

By default, when an action is called, Wheels will look for a view file with the same name as the action. It then hands off the processing to the view to display the user interface. In our case, Wheels tried to find a view file for our `say/hello` action and couldn't find one.

Let's remedy the situation and create a view file. View files are simple CFML pages that handle the output of our application. In most cases, views will return HTML code to the browser. By default, the view files will have the same name as our controller actions and will be grouped into a directory under the `view` directory. This new directory will have the same name as our controller.

Find the `views` directory in the root of your Wheels installation. There will be a few directories in there already. For now, we need to create a new directory in the `views` directory called `say`. This is the same name as the controller that we created above.

Now inside the `say` directory, create a file called `hello.cfm`. In the `hello.cfm` file, add the following line of code:

```
<h1>Hello World</h1>
```

Save your `hello.cfm` file, and let's call our `say/hello` action once again. You have your first working Wheels page if your browser looks like *Figure 3* below.

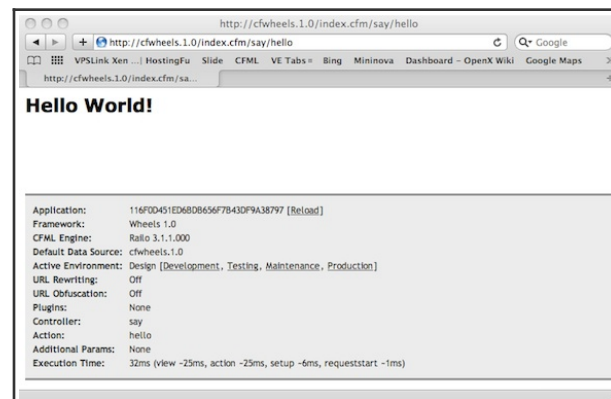


Figure 3: Your first working Wheels action.

You have just created your first functional Wheels page, albeit it is a very simple one. Pat yourself on the back, go grab a snack, and when you're ready, let's go on and extend the functionality of our *Hello World!* application a little more.

Adding Dynamic Content to Your View

We will add some simple dynamic content to our `hello` action and add a second action to the application. We'll then use some Wheels code to tie the 2 actions together. Let's get get to it!

The Dynamic Content

The first thing we are going to do is to add some dynamic content to our `say/hello` action. Modify your `say` controller so it looks like the code block below:

```
<cfcomponent extends="Controller">
    <cffunction name="hello">
        <cfset time = Now()>
    </cffunction>
</cfcomponent>
```

All we are doing here is creating a variable called `time` and setting its value to the current server time using the basic ColdFusion `Now()` function. When we do this, the variable becomes immediately available to our view code.

Why not just set up this value directly in the view? If you think about it, maybe the logic behind the value of `time` may eventually change. What if eventually we want to display its value based on the user's time zone? What if later we decide to pull it from a web service instead? Remember, the controller is supposed to coordinate all of the data and business logic, not the view.

Displaying the Dynamic Content

Next, we will modify our `say/hello.cfm` view file so that it looks like the code block below. When we do this, the value will be displayed in the browser.

```
<h1>Hello World</h1>
<p>Current time:<cfoutput#time#</cfoutput*>/p>
```

Now call your `say/hello` action again in your browser. Your browser should look like *Figure 4* below.

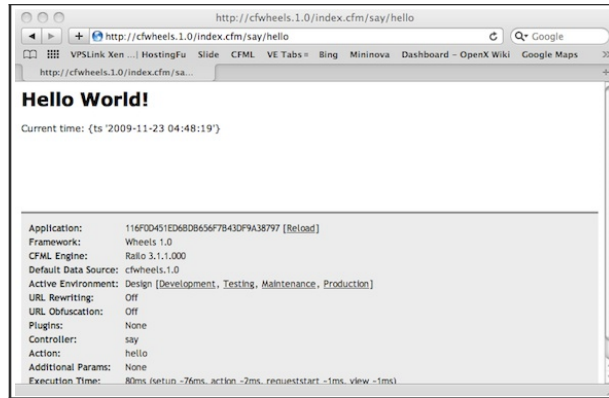


Figure 4: Hello World with the current date and time.

This simple example showed that any dynamic content created in a controller action is available to the corresponding view file. In our application, we created a `time` variable in the `say/hello` controller action and display that variable in our `say/hello.cfm` view file.

Adding a Second Action: Goodbye

Now we will expand the functionality of our application once again by adding a second action to our `say` controller. If you feel adventurous, go ahead and add a `goodbye` action to the `say` controller on your own, then create a `goodbye.cfm` view file that displays a "Goodbye" message to the user. If you're not feeling that adventurous, we'll quickly go step by step.

First, modify the `say` controller file so that it looks like the code block below.

```
<cfcomponent extends="Controller">
    <cffunction name="hello">
        <cfset time = Now()>
    </cffunction>

    <cffunction name="goodbye"></cffunction>
</cfcomponent>
```

Now go to the `view/say` directory and create a `goodbye.cfm` page.

Add the following code to the `goodbye.cfm` page and save it.

```
<h1>Goodbye World</h1>
```

If we did everything right, we should be able to call the new `say/goodbye` action using the following URL:

```
http://cfwheels.1.0/index.cfm/say/goodbye
```

Your browser should look like *Figure 5* below:

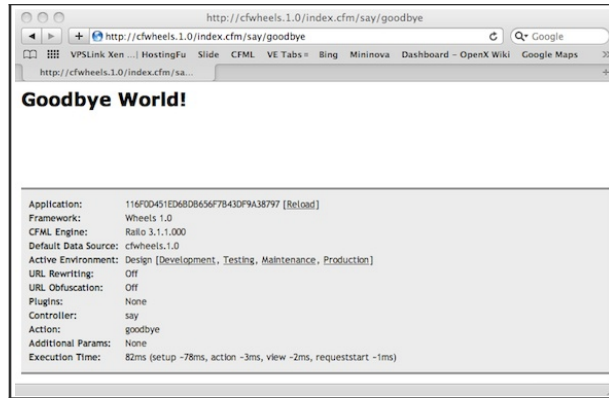


Figure 3: Your new goodbye action.

Linking to Other Actions

Now let's link our two actions together. We will do this by adding a link to the bottom of each page so that it calls the other page.

Linking Hello to Goodbye

Open the `say/hello.cfm` view file. We are going to add a line of code to the end of this file so our `say/hello.cfm` view file looks like the code block below:

```
<h1>Hello World</h1>  
<p>Current time:<cfoutput>#time#</cfoutput></p>  
<p>Time to say<cfoutput>#linkTo(text="goodbye", action="goodbye")</cfoutput></p>
```

The `linkTo()` function is a built-in Wheels function. In this case, we are passing 2 named parameters to it. The first parameter, `text`, is the text that will be displayed in the hyperlink. The second parameter, `action`, defines the action to point the link to. By using this built-in function, your application's mainURL may change, and even controllers and actions may get shifted around, but you won't suffer from the dreaded dead link. Wheels will always create a valid link for you as long as you configure it correctly when you make infrastructure changes to your application.

Once you have added the additional line of code to the end of the `say/hello.cfm` view file, save your file and call the `say/hello` action from your browser. Your browser should look like *Figure 6* below.

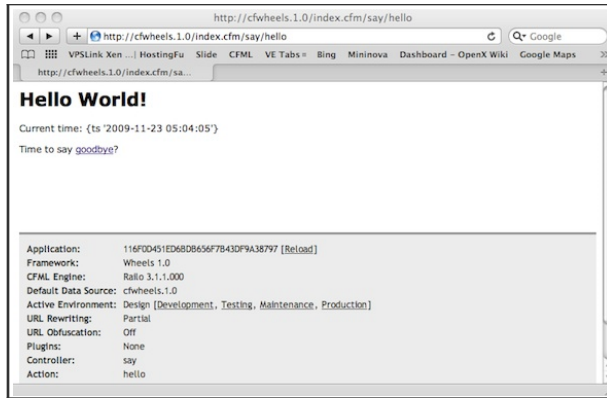


Figure 6: Your say/hello action with a link to the goodbye action.

You can see that Wheels created a link for us and added an appropriate URL for the say/goodbye action to the link.

Linking Goodbye to Hello

Let's complete our little app and add a corresponding link to the bottom of our say/goodbye.cfm view page.

Open your say/goodbye.cfm view page and modify it so it looks like the code block below.

```
<h1>Goodbye World</h1>
<p>Time to say <cfoutput>#linkTo(text="hello", action="hello")</cfoutput>#</p>
```

If you now call the say/goodbye action in your browser, your browser should look like Figure 7 below.

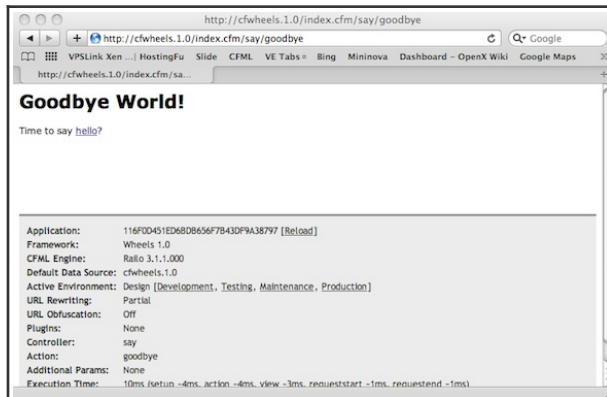


Figure 7: Your say/goodbye action with a link to the hello action.

Much More to Learn

You now know enough to be dangerous with ColdFusion on Wheels. Look out! But there are many more powerful features to cover. You may have noticed that we haven't even talked about the M in MVC.

No worries. We will get there. And we think you will enjoy it.

Beginner Tutorial: Hello Database

A quick tutorial that demonstrates how quickly you can get database connectivity up and running with Wheels.

Wheels's built in model provides your application with some simple and powerful functionality for interacting with databases. To get started, you make some simple configurations, call some functions within your controllers, and that's it. Best yet, you will rarely ever need to write SQL code to get those redundant CRUD tasks out of the way.

Our Sample Application: User Management

We'll learn by building part of a sample user management application. This tutorial will teach you the basics of interacting with Wheels'sORM.

Setting up the Data Source

By default, Wheels will connect to a data source that has the same name as the folder containing your Wheels application. So if your Wheels application is in a folder called `C:\websites\mysite\blog\`, then it will connect to a data source named `blog`.

To change this default behavior, open the file at `config/settings.cfm`. In a fresh install of Wheels, you'll see some commented-out lines of code that read as such:

```
<!---
    If you leave these settings commented out, Wheels will set the data source name
    to the same name as the folder the application resides in.
    <cfset set(dataSourceName="")>
    <cfset set(dataSourceUserName="")>
    <cfset set(dataSourcePassword="")>
-->
```

Uncomment the lines that tell Wheels what it needs to know about the data source and provide the appropriate values. This may include values for `dataSourceName`, `dataSourceUserName`, and `dataSourcePassword`.

```
<cfset set(dataSourceName="myblogapp")>
<cfset set(dataSourceUserName="marty")>
<cfset set(dataSourcePassword="cfly")>
```

Our Sample Data Structure

Wheels supports MySQL, SQL Server, PostgreSQL, and Oracle. It doesn't matter which DBMS you use. We will all be writing the same CFML code to interact with the database. Wheels does everything behind the scenes that needs to be done to work with each DBMS.

That said, here's a quick look at a table that you'll need in your database:

Table: users

Column Name	Data Type	Extra
id	int	auto increment, primary key
name	varchar(100)	
email	varchar(255)	
password	varchar(15)	

Note a couple things about these tables:

1. The table name is plural.
2. The table has an auto-incrementing primary key named id.

These are database *conventions* used by Wheels. This framework strongly encourages that everyone follow convention over configuration. That way everyone is doing things mostly the same way, leading to less maintenance and training headaches down the road.

Fortunately, there are ways of going outside of these conventions when you really need it. But let's learn the conventional way first. Sometimes you need to learn the rules before you can know how to break them, cowboy.

Adding Users

First, let's create a simple form for adding a new user to the `users` table. To do this, we will use Wheels's *form helper* functions. Wheels includes a whole range of functions that simplifies all of the tasks that you need to display forms and communicate errors to the user.

Creating the Form

Now create a new file in `views/users` called `new.cfm`. This will contain the view code for our simple form.

Next, add these lines of code to the new file:

```
<cfoutput>
<h1>Create a New User</h1>
#startFormTag(action="create")#
    <div>#textField(objectName="user", property="name", label="Name")#>
    <div>#textField(objectName="user", property="email", label="Email")#>
    <div>#passwordField(objectName="user", property="password", label="Password")#>
    <div>#submitTag()#/div>
```

```
#endFormTag()#  
</cfoutput>
```

Form Helpers

What we've done here is use form helpers to generate all of the form fields necessary for creating a new user in our database. It may feel a little strange using functions to generate form elements, but it will soon become clear why we're doing this. Trust us on this one... You'll love it!

To generate the form tag's `action` attribute, the `startFormTag()` function takes parameters similar to the `linkTo()` function that we introduced in the [Beginner Tutorial: Hello World](#) tutorial. We can pass in `controller`, `action`, `key`, and other route- and parameter-defined URLs just like we do with `linkTo()`.

To end the form, we use the `endFormTag()` function. Easy peasy, lemon squeezy.

The `textField()` and `passwordField()` helpers are similar. As you probably guessed, they create `<input>` elements with `type="text"` and `type="password"`, respectively. And the `submitTag()` function creates an `<input type="submit" />` element.

One thing you'll notice is the `textField()` and `passwordField()` functions accept arguments called `objectName` and `property`. As it turns out, this particular view code will throw an error because these functions are expecting an object named `user`. Let's fix that.

Supplying the Form with Data

We need to supply our view code with an object called `user`. Because the controller is responsible for providing the views with data, we'll set it there.

Create a new ColdFusion component at `controllers/Users.cfc`. If you're using a case sensitive operating system like UNIX or Linux, then it is important to name the CFC with a capital letter.

As it turns out, our controller needs to provide the view with a blank `user` object (whose instance variable will also be called `user` in this case). In our new action, we will use the `model()` function to generate a new instance of the `user` model.

To get a blank set of properties in the model, we'll also call the generated model's `new()` method.

```
<cfcomponent extends="Controller">  
    <cffunction name="new">  
        <cfset user = model("user").new(>  
    </cffunction>  
</cfcomponent>
```

Wheels will automatically know that we're talking about the `users` database table when we instantiate a `user` model. The convention: database tables are plural and their corresponding Wheels models are singular.

Why is our model name singular instead of plural? When we're talking about a single record in the `users` database, we represent that with a model object. So the `users` table contains many `user` objects. It just works better in conversation.

The Generated Form

Now when we run the URL at `http://localhost/users/new`, we'll see the form with the fields that we defined.

The HTML generated by your application will look something like this:

```
<h1>Create a New User</h1>
<form action="/users/create"method="post">
  <div>
    <label for="user-name">Name
    <input id="user-name" type="text" value="" name="user[name]" /></label>
  </div>
  <div>
    <label for="user-email">Email
    <input id="user-email" type="text" value="" name="user[email]" /></label>
  </div>
  <div>
    <label for="user-password">Password
    <input id="user-password" type="password" value="" name="user[password]" /></label>
  </div>
  <div><input value="Save changes" type="submit" /></div>
</form>
```

So far we have a fairly well-formed, accessible form, without writing a bunch of repetitive markup.

Handling the Form Submission

Next, we'll code the `create` action in the controller to handle the form submission and save the new user to the database.

A basic way of doing this is using the model object's `create()` method:

```
<cfunction name="create">
  <cfset user = model("user").create(params.user)>
  <cfset redirectTo(
    action="index",
    success="User #user.name# created successfully."
  )>
</cfunction>
```

Because we used the `objectName` argument in the fields of our form, we can access the data as a struct in the `params` struct.

There are more things that we can do in the `create` action to handle validation, but let's keep it simple in this tutorial.

Listing Users

Notice that our `create` action above redirects the user to the `index` action using the `redirectTo()` function. We'll use this action to list all users in the system with "Edit" links. We'll also provide a link to the create form that we just coded.

First, let's get the data that the listing needs. Create an action named `index` in the `users` controller like so:

```
<cffunction name=index>
    <cfset users = model("user").findAll(order=name)>
</cffunction>
```

This call to the model's `findAll()` method will return a query object of all users in the system. By using the method's `order` argument, we're also telling the database to order the records by `name`.

In the view at `views/users/index.cfm`, it's as simple as looping through the query and outputting the data.

```
<cfoutput>
<h1>Users</h1>
#flashMessages("success")#
<p>#linkTo(text="+ Add New User", action="new")#
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Email</th>
      <th colspan="2"></th>
    </tr>
  </thead>
  <tbody>
    <cfloop query=users>
      <tr>
        <td>#users.name#/td>
        <td>#users.email#/td>
        <td>
          #linkTo(
            text="Edit", action="edit", key=users.id,
            title="Edit #users.name#"
          )#
        </td>
        <td>
          #linkTo(
            text="Delete", action="delete", key=users.id,
            title="Delete #users.name#",
            confirm="Are you sure that you want to delete
              #users.name#?"
          )#
        </td>
      </tr>
    </cfloop>
  </tbody>
</table>
```

```

        </tr>
      </tbody>
    </table>
  </cfoutput>

```

Editing Users

We'll now show another cool aspect of form helpers by creating a screen for editing users.

Coding the Edit Form

You probably noticed in the code listed above that we'll have an action for editing users called `edit`. This action expects a key as well which is passed in the URL by default.

Given the provided `key`, we'll have the action load the appropriate `user` object to pass on to the view:

```

<cffunction name="edit">
    <cfset user = model("user").findByKey(params.key)>
</cffunction>

```

The view at `views/user/edit.cfm` looks almost exactly the same as the view for creating a user:

```

<cfoutput>
<h1>Edit User #user.name</h1>
<cfif flashKeyExists("success")>
    <p class="success">#flash("success")</p>
</cfif>
#startFormTag(action="update")#
    <div>#hiddenField(objectName="user", property="id")#</div>
    <div>#textField(objectName="user", property="name", label="Name")#</div>
    <div>#textField(objectName="user", property="email", label="Email")#</div>
    <div>
        #passwordField(objectName="user", property="password", label="Password")#
    </div>
    <div>#submitTag()#</div>
#endFormTag()#
</cfoutput>

```

But an interesting thing happens. Because the form fields are bound to the `user` object via the form helpers' `objectName` arguments, the form will automatically provide default values based on the object's properties.

With the `user` model populated, we'll end up seeing code similar to this:

```
<h1>Edit User Homer Simpson</h1>
<form action="/users/update" method="post">
  <div><input type="hidden" name="user[id]" value="15" /></div>
  <div>
    <label for="user-name">Name
    <input
      id="user-name" type="text" value="Homer Simpson"
      name="user[name]" /></label>
  </div>
  <div>
    <label for="user-email">Email
    <input
      id="user-email" type="text" value="homerj@nuclearpower.com"
      name="user[email]" /></label>
  </div>
  <div>
    <label for="user-password">Password
    <input
      id="user-password" type="password" value="donuts.mmm"
      name="user[password]" /></label>
  </div>
  <div><input value="Save changes" type="submit" /></div>
</form>
```

Pretty cool, huh?

Opportunities for Refactoring

There's a lot of repetition in the `add` and `edit` forms. You'd imagine that we could factor out most of this code into a single view file. To keep this tutorial from becoming a book, we'll just continue on knowing that this could be better.

Handling the Edit Form Submission

Now we'll create the `update` action. This will be similar to the `create` action, except it will be updating the user object:

```
<cffunction name="update">
  <cfset user = model("user").findByKey(params.user.id)
  <cfset user.update(params.user)
  <cfset redirectTo(
    action="edit",
```

```
        key=user.id
        success="User #user.name# updated successfully."
    }>
</cffunction>
```

To update the user, simply call its `update()` method with the `user` struct passed from the form via `params`. It's that simple.

After the update, we'll add a success message using the Flash and send the end user back to the edit form in case they want to make more changes.

Deleting Users

Notice in our listing above that we have a `delete` action. Our `linkTo()` call has an argument named `confirm`, which will include some simple JavaScript to pop up a box asking the end user to confirm the deletion before running the `delete` action.

Here's what our `delete` action would look like:

```
<cffunction name="delete">
    <cfset user = model("user").findByKey(params.key)>
    <cfset user.delete()>
    <cfset redirectTo(
        action="index",
        success="#user.name# was successfully deleted."
    )>
</cffunction>
```

We simply load the user using the model's `findByKey()` method and then call the object's `delete()` method. That's all there is to it.

Database Says Hello

We've shown you quite a few of the basics in getting a simple user database up and running. We hope that this has whet your appetite to see some of the power packed into the ColdFusion on Wheels framework. There's plenty more.

Be sure to read on to some of the related chapters listed below to learn more about working with Wheels's ORM.

Tutorial: Wheels, AJAX, and You

Using Wheels to develop web applications with AJAX features is a breeze. You have several options and tools at your disposal, which we'll cover in this chapter.

Wheels was designed to be as lightweight as possible, so this keeps your options fairly open for developing AJAX features into your application. We will cover two different approaches in this chapter:

1. "Do it yourself" method with a fresh out-of-the box install of Wheels
2. "Plugins" method with the help of a couple different plugins built by community members

While there are several flavors of JavaScript libraries out there with AJAX support, we will be using the jQuery framework in this tutorial. Let's assume that you are fairly familiar with the basics of jQuery and know how to set it up.

For this tutorial, let's create the simplest example of all: a link that will render a message back to the user without refreshing the page.

Approach #1: "Do It Yourself"

In this example, we'll wire up some simple JavaScript code that calls a Wheels action asynchronously. All of this will be done with basic jQuery code and built-in Wheels functionality.

First, let's create a link to a controller's action in a view file, like so:

```
<cfoutput>
<!-- View code -->
<h1>/h1>
<p>/p>

#linkTo(text="Alert me!", controller="say", action="hello", id="alert-button")#
</cfoutput>
```

That piece of code by itself will work just like you expect it to. When you click the link, you will load the `hello` action inside the `say` controller.

But let's make it into an asynchronous request. Add this JavaScript (either on the page inside `script` tags or in a separate `.js` file included via `javascriptIncludeTag()`):

```
(function($){$(document).ready(function(){
    // Listen to the "click" event of the "alert-button" link and make an AJAX request
    $("#alert-button").click(function(){
        $.ajax{
            type: "POST",
```

```

        url: $(this).attr("href") + "?format=json", // References "/say/hello?format=json"
        dataType: "json",
        success: function(response)
            $("h1").html(response.message);
            $("p").html(response.time);
        }
    });
    return false; // keeps the normal request from firing
});
})(jQuery);

```

With that code, we are listening to the `click` event of the hyperlink, which will make an asynchronous request to the `hello` action in the `say` controller. Additionally, the JavaScript call is passing a URL parameter called `format` set to `json`.

Note that the `success` block inserts keys from the `response` into the empty `h1` and `p` blocks in the calling view. (You may have been wondering about those when you saw the first example. Mystery solved.)

The last thing that we need to do is implement the `say/hello` action. Note that the request expects a `dataType` of `JSON`. By default, Wheels controllers only generate HTML responses, but there is an easy way to generate JSON instead using Wheels's `provides()` and `renderWith()` functions:

```

<!-- Controller code -->
<cffunction name="init">
    <cfset provides("html,json")>
</cffunction>

<cffunction name="hello">
    <!-- Prepare the message for the user -->
    <cfset greeting = {}>
    <cfset greeting["message"] = "Hi there">
    <cfset greeting["time"] = Now()>

    <!-- Respond to all requests with `renderWith()` -->
    <cfset renderWith(greeting)>
</cffunction>

```

In this controller's `init()` method, we use the `provides()` function to indicate that we want all actions in the controller to be able to respond with the data in HTML or JSON formats. Note that the client calling the action can request the type by passing a URL parameter named `format` or by sending the format in the request header.

The call to `renderWith()` in the `hello` action takes care of the translation to the requested format. Our JavaScript is requesting JSON so Wheels will format the `greeting` struct as JSON automatically and send it back to the client. If the client requested HTML or the default of none, Wheels will process and serve the view template at `views/say/hello.cfm`. For more information about `provides()` and `renderWith()`, reference the chapter on Responding with Multiple Formats.

Lastly, notice the lines where we're setting `greeting["message"]` and `greeting["time"]`. Due to the case-insensitive nature of ColdFusion, we recommend setting variables to be consumed by JavaScript using bracket notation like that. If you do not use that notation (i.e., `greetings.message` and `greetings.time` instead), your JavaScript will need to reference those keys from the JSON as `MESSAGE` and `TIME` (all caps). Unless you like turning caps lock on and off, you can see how that would get annoying after some time.

Assuming you already included jQuery in your application and you followed the code examples above, you now have a simple AJAX-powered web application built on Wheels. After clicking that `Alert me!` link, your `say` controller will respond back to you the

serialized message via AJAX. jQuery will parse the JSON object and populate the `h1` and `p` with the appropriate data.

Approach #2: Remote Form Helpers Plugin

The Remote Form Helpers adds a set of CFMJS functions (CFML + JavaScript) as well as a set of new form helpers that work exclusively with AJAX.

Remote Form Helpers Plugin

By itself, the Remote Form Helpers plugin will add another type of response to your application: the `javascript` response. Let's replicate the previous example using Remote Form Helpers. (Refer to the plugin's documentation and our chapter on Using and Creating Plugins for more information.)

First, let's replace the `linkTo()` call from the first example with a call to a function included by the plugin, called `remoteLinkTo()`:

```
<cfoutput>

<!-- View code -->
<h1></h1>
<p></p>

#remoteLinkTo(text="Alert me!", controller="say", action="hello")#

</cfoutput>
```

`remoteLinkTo()` is all you need in your view now, so go ahead and remove the whole `script` block from the first example. This new function will take care of creating the JavaScript for you.

Now let's review that `say` controller:

```
<!-- Controller code -->
<cffunction name="hello">
  <!-- Prepare the message for the user -->
  <cfset greeting = {}>
  <cfset greeting["message"] = "Hi there">
  <cfset greeting["time"] = Now()>

  <!-- If your request is an AJAX request, respond with the `views/say/hello.js.cfm remote view` -->
  <cfif isAjax()>
    <cfset renderRemotePage(>)
  </cfif>

  <!-- Otherwise, let Wheels do its normal HTML response (the `views/say/hello.cfm` view file) -->
</cffunction>
```

Wait, what? `hello.js.cfm`? Yes, the Remote Form Helpers plugin enables you to use a whole new set of view files. All files that have a `.js.cfm` extension are considered "remote view files" and will be used with your `renderRemotePage()` function to send it back to your initial request.

Now let's create the file at `views/say/hello.js.cfm` and add this code:

```
<!--- Remote View code --->
<cfoutput>

#pageInsertHTML(selector="h1", content=greeting.message)#
#pageInsertHTML(selector="p", content=greeting.time)#

</cfoutput>
```

That's it. Without a single line of JavaScript, we were able to setup an AJAX-powered Wheels application.

AJAX in Wheels Explained

That is it! Hopefully now you have a clearer picture on how to create AJAX-based features for your web applications.

Remember that any of the 3 examples work just fine. You have several ways to accomplish the same goal. Choose the one that fits your style and needs.

Conventions

With a convention-over-configuration framework like Wheels, it's important to know these conventions. This is your guide.

There is a specific set of standards that Wheels follows when you run it in its default state. This is to save you time. With conventions in place, you can get started coding without worrying about configuring every little detail.

But it is important for you to know these conventions, especially if you're running an operating system and/or DBMS configuration that's picky about things like case sensitivity.

URLs

Wheels uses a very flexible routing system to match your application's URLs to controllers, views, and parameters.

Within this routing system is a *default route* that handles many scenarios that you'll run across as a developer. The default route is mapped using the pattern `[controller]/[action]/[key]`.

Consider this example URL:

■ `http://localhost/users/edit/12`

This maps to the `user` controller, `edit` action, and a key of `12`. For all intents and purposes, this will load a view for editing a user with a primary key value in the database of `12`.

This URL pattern works up the chain and will also handle the following example URLs:

URL	Controller	Action	Key
<code>http://localhost/users/edit/12</code>	<code>users</code>	<code>edit</code>	<code>12</code>
<code>http://localhost/users/add</code>	<code>users</code>	<code>add</code>	<code>Undefined</code>
<code>http://localhost/users</code>	<code>users</code>	<code>index</code>	<code>Undefined</code>

See [Using Routes](#) for instructions on overriding this behavior.

Naming Conventions for Controllers, Actions, and Views

Controllers, actions, and views are closely linked together by default. And how you name them will influence the URLs that Wheels will generate.

Controllers

First, a controller is a CFC file placed in the `controllers` folder. It should be named in `PascalCase`. For example, a site map controller would be stored at `controllers/SiteMap.cfc`.

Multi-word controllers will be delimited by hyphens in their calling URLs. For example, a URL of `/site-map` will reference the `SiteMap` controller.

See [Using Routes](#) for instructions on overriding this behavior.

Actions

Methods within the controllers, known as actions, should be named in `camelCase`.

Like with controllers, any time a capital letter is used in `camelCase`, a hyphen will be used as a word delimiter in the corresponding URL. For example, a URL of `/site-map/search-engines` will reference the `searchEngines` action in the `SiteMap` controller.

See [Using Routes](#) for instructions on overriding this behavior.

Views

By default, view files are named after the action names and are stored in folders that correspond to controller names. Both the folder names and view file names should be all lowercase, and there is no word delimiter.

In our `/site-map/search-engines` URL example, the corresponding view file would be stored at `views/sitemap/searchengines.cfm`.

For information on overriding this behavior, refer to documentation for the `renderPage()` function and read the [Pages](#) chapter.

Layouts

A special type of view file called a *layout* defines markup that should surround the views loaded by the application. The default layout is stored at `views/layout.cfm` and is automatically used by all views in the application.

Controller-level layouts can also be set automatically by creating a file called `layout.cfm` and storing it in the given controller's view folder. For example, to create a layout for the `users` controller, the file would be stored at `views/users/layout.cfm`.

When a controller-level layout is present, it overrides the default layout stored in the root `views` folder.

For information on overriding the layout file to be loaded by an action, see the chapter on [Using Layouts](#) and documentation for the `renderPage()` function.

Naming Conventions for Models and Databases

By default, the names of `Wheels` models, model properties, database tables, and database fields all relate to each other. `Wheels` even sets a sensible default for the CFML data source used for database interactions.

Data Sources

Wheels will automatically look for a data source with the same name as the folder that your application is deployed in. If your Wheels application is in a folder called `blog`, Wheels will look for a data source called `blog`, for example.

Refer to the Configuration and Defaults chapter for instructions on overriding data source information.

Plural Database Table Names, Singular Model Names

Wheels adopts a Rails-style naming conventions for database tables and model files. Think of a database table as a collection of model objects; therefore, it is named with a plural name. Think of a model object as a representation of a single record from the database table; therefore, it is named with a singular word.

For example, a `user` model represents a record from the `users` database table. Wheels also recognizes plural patterns like `binary/binaries`, `mouse/mice`, `child/children`, etc.

Like controller files, models are also CFCs and are named in `PascalCase`. They are stored in the `models` folder. So the `user` model would be stored at `models/User.cfc`.

For instructions on overriding database naming conventions, refer to documentation for the `table()` function and the chapter on Object Relational Mapping.

Everything in the Database Is Lowercase

In your database, both table names and column names should be lowercase. The `customersegments` table could have fields called `title`, `regionid`, and `incomelevel`, for example.

Because of CFML's case insensitive nature, we recommend that you refer to model names and corresponding properties in `camelCase`. This makes for easier readability in your application code.

In the `customersegments` example above, you could refer to the properties in your CFML as `title`, `regionId`, and `incomeLevel` to stick to CFML's Java-style roots. (Built-in CFML functions are often written in `camelCase` and `PascalCase`, after all.)

For information on overriding column and property names, refer to documentation for the `property()` function and the Object Relational Mapping chapter.

Configuration and Defaults

There are many default values and settings that you can tweak in Wheels when you need to. Some of them are conventions and others are just configurations available for you to change. You can even change argument defaults for built-in Wheels functions to keep your code DRYer.

For more details on what you can configure, read the Configuration and Defaults chapter.

Configuration and Defaults

An overview of Wheels configuration and how is it used in your applications. Learn how to override a Wheels convention to make it your own.

We all love the "Convention over Configuration" motto of Wheels, but what about those two cases that pop into everyone's head? *What if I want to develop in my own way?* Or, *What about an existing application that I need to port into Wheels?* Gladly, that's what configuration and defaults are there for. Let's take a look at exactly how is this performed.

Where Configurations Happen

You will find configuration files in the `config` folder of your Wheels application. In general, most of your settings will go in `config/settings.cfm`

You can also set values based on what environment you have set. For example, you can have different values for your settings depending on whether you're in `design` mode or `production` mode. See the chapter on Switching Environments for more details.

How to Set Configurations

To change a Wheels application default, you generally use the `set()` function. With it, you can perform all sorts of tweaks to the framework's default behaviors.

How to Access Configuration Values

Use the `get()` function to access the value of a Wheels application setting. Just pass it the name of the setting.

```
<cfif get("environment") is "production">
  <!--- Do something for production environment --->
</cfif>
```

Setting CFML application Configurations

In CFML's standard `Application.cfc`, you can normally set values for your application's properties in the `this` scope. Wheels still provides these options to you in the file at `config/app.cfm`.

Here is an example of what can go in `config/app.cfm`:

```
<cfset this.name = "TheNextSiteToBeatTwitter">
<cfset this.sessionManagement = false>
<cfset this.customTagPaths &listAppend
```

```
this.customTagPaths,  
ExpandPath("../customtags")  
)>
```

Types of Configurations Available

There are several types of configurations that you can perform in Wheels to override all those default behaviors. In Wheels, you can find all these configuration options:

- URL rewrite settings
- Data source settings
- Environment settings
- Caching settings
- Function settings
- Miscellaneous settings

Let's take a closer look at each of these options.

URL Rewrite Settings

Sometimes it is useful for our applications to "force" URL rewriting. By default, Wheels will try to determinate what type of URL rewriting to perform and set it up for you. But you can force in or out this setting by using the example below:

```
<cfset set(urlRewriting="Off")>
```

The code above will tell Wheels to skip its automatic detection of the URL Rewriting capabilities and just set it as "Off".

You can also set it to "Partial" if you believe that your web server is capable of rewriting the URL as folders after `index.cfm`.

For more information, read the chapter about URL Rewriting.

Data Source Settings

Probably the most important configuration of them all. What is an application without a database to store all of its precious data?

The data source configuration is what tells Wheels which database to use for all of its models. (This can be overridden on a per model basis, but that will be covered later.) To set this up in Wheels, it's just as easy as the previous example:

```
<cfset set(dataSourceName=yourDataSourceName)>  
<cfset set(dataSourceUserName=yourDataSourceUsername)>  
<cfset set(dataSourcePassword=yourDataSourcePassword)>
```

Debugging and Error Settings

Not only are the environments useful for separating your production settings from your "under development" settings, but they are also opportunities for you to override settings that will only take effect in a specified environment.

For example, let's say that we want to enable debugging information in our "development" environment temporarily:

```
<!-- /config/development/settings.cfm -->  
<cfset set(showDebugInformation=false)>
```

Full Listing of Environment Settings

Name	Type	Default	Description
errorEmailServer	string	[empty string]	Server to use to send out error emails. When left blank, this defaults to settings in the ColdFusion Administrator (if set).
errorEmailAddress	string	[empty string]	Comma-delimited list of email address to send error notifications to. Only applies if <code>sendEmailOnError</code> is set to true.
errorEmailSubject	string	Error	Subject of email that gets sent to administrators on errors. Only applies if <code>sendEmailOnError</code> is set to true.
excludeFromErrorEmail	string	[empty string]	List of variables (or entire scopes) to exclude from the scope dumps included in error emails. Use this to keep sensitive information from being sent in plain text over email.
sendEmailOnError	boolean	Enabled in production environments that have a TLD like .com, .org, etc.	When set to true, Wheels will send an email to administrators whenever Wheels throws an error.
showDebugInformation	boolean	Enabled in design and development	When set to true, Wheels will show debugging information in the footers of your pages.
showErrorInformation	boolean	Enabled in design, development, maintenance, and testing	When set to false, Wheels will run and display code stored at <code>events/onerror.cfm</code> instead of revealing CFML errors.

For more information, refer to the chapter about Switching Environments.

Caching Settings

Wheels does a pretty good job at caching the framework and its output to speed up your application. But if personalization is key in your application, finer control over caching settings will become more important.

Let's say your application generates dynamic routes and you need it to check the routes on each request. This task will be as simple as this line of code:

```
<cfset set(cacheRoutes=false)>
```

Full Listing of Caching Settings

Name	Type	Default	Description
cacheActions	boolean	Enabled in maintenance, testing, and production	When set to true, Wheels will cache output generated by actions when specified (in a <code>caches()</code> call, for example).
cacheControllerInitialization	boolean	Enabled in development, maintenance, testing, and production	When set to false, any changes you make to the <code>init()</code> function in the controller file will be picked up immediately.
cacheCullInterval	numeric	5	Number of minutes between each culling action. The reason the cache is not culled during each request is to keep performance as high as possible.
cacheCullPercentage	numeric	10	If you set this value to 10, then at most, 10% of expired items will be deleted from the cache.
cacheDatabaseSchema	boolean	Enabled in development, maintenance, testing, and production	When set to false, you can add a field to the database, and Wheels will pick that up right away.
cacheFileChecking	boolean	Enabled in development, maintenance, testing, and production	When set to true, Wheels will cache whether or not controller, helper, and layout files exist
cacheImages	boolean	Enabled in development, maintenance, testing, and production	When set to true, Wheels will cache general image information used in <code>imageTag()</code> like width and height.
cacheModelInitialization	boolean	Enabled in development, maintenance, testing, and production	When set to false, any changes you make to the <code>init()</code> function in the model file will be picked up immediately.
cachePages	boolean	Enabled in maintenance, testing, and production	When set to true, Wheels will cache output generated by a view page when specified (in a <code>renderPage()</code> call, for example).
cachePartials	boolean	Enabled in maintenance, testing, and production	When set to true, Wheels will cache output generated by partials when specified (in a <code>includePartial()</code> call for example).
cacheQueries	boolean	Enabled in maintenance, testing, and production	When set to true, Wheels will cache SQL queries when specified (in a <code>findAll()</code> call, for example).
clearQueryCacheOnReload	boolean	true	Set to true to clear any queries that Wheels has cached on application reload.
cacheRoutes	boolean	Enabled in development, maintenance, testing, and production	When set to true, Wheels will cache routes across all pageviews.

defaultCacheTime	numeric	60	Number of minutes an item should be cached when it has not been specifically set through one of the functions that perform the caching in Wheels (i.e., <code>cache()</code> , <code>findAll()</code> , <code>includePartial()</code> , etc.)
maximumItemsToCache	numeric	5000	Maximum number of items to store in Wheels's cache at one time. When the cache is full, items will be deleted automatically at a regular interval based on the other cache settings.

For more information, refer to the chapter on Caching.

Function Settings

OK, here it's where the fun begins! Wheels includes a lot of functions to make your life as a CFML developer easier. A lot of those functions have sensible default argument values to minimize the amount of code that you need to write. And yes, you guessed it, Wheels lets you override those default argument values application-wide.

Let's look at a little of example:

```
<cfset set(functionName="findAll", perPage=20)>
```

That little line of code will make all calls to the `findAll()` method in Wheels return a maximum number of 20 records per page (if pagination is enabled for that `findAll()` call). How great is that? You don't need to set the `perPage` value for every single call to `findAll()` if you have a different requirement than the Wheels default of 10 records.

ORM Settings

The Wheels ORM provides many sensible conventions and defaults, but sometimes your data structure requires different column naming or behavior than what Wheels expects out of the box. Use these settings to change those naming conventions or behaviors across your entire application.

For example, if we wanted to prefix all of the database table names in our application with `blog_` but didn't want to include that at the beginning of model names, we would do this:

```
<cfset set(tableNamePrefix="blog_")>
```

Now your `post` model will map to the `blog_posts` table, `comment` model will map to the `blog_comments` table, etc.

Full Listing of ORM Settings

Name	Type	Default	Description
------	------	---------	-------------

<code>afterFindCallbackLegacySupport</code>	<code>boolean</code>	<code>true</code>	When this is set to <code>false</code> and you're implementing an <code>afterFind()</code> callback, you need to write the same logic for both the <code>this</code> scope (for objects) and <code>arguments</code> scope (for queries). Setting this to <code>false</code> makes both ways use the <code>arguments</code> scope so you don't need to duplicate logic. Note that the default is <code>true</code> for backwards compatibility.
<code>automaticValidations</code>	<code>boolean</code>	<code>true</code>	Set to <code>false</code> to stop Wheels from automatically running object validations based on column settings in your database.
<code>setUpdatedAtOnCreate</code>	<code>boolean</code>	<code>true</code>	Set to <code>false</code> to stop Wheels from populating the <code>updatedAt</code> timestamp with the <code>createdAt</code> timestamp's value.
<code>softDeleteProperty</code>	<code>string</code>	<code>deletedAt</code>	Name of database column that Wheels will look for in order to enforce soft deletes.
<code>tableNamePrefix</code>	<code>string</code>	[empty string]	String to prefix all database tables with so you don't need to define your model objects including it. Useful in environments that have table naming conventions like starting all table names with <code>tbl</code>
<code>timeStampOnCreateProperty</code>	<code>string</code>	<code>createdAt</code>	Name of database column that Wheels will look for in order to automatically store a "created at" time stamp when records are created.
<code>timeStampOnUpdateProperty</code>	<code>string</code>	<code>updatedAt</code>	Name of the database column that Wheels will look for in order to automatically store an "updated at" time stamp when records are updated.
<code>transactionMode</code>	<code>string</code>	<code>commit</code>	Use <code>commit</code> , <code>rollback</code> , or <code>none</code> to set default transaction handling for creates, updates and deletes.
<code>useExpandedColumnAliases</code>	<code>boolean</code>	<code>true</code>	When set to <code>true</code> , Wheels will always prepend children objects' names to columns included via <code>findAll()</code> 's <code>include</code> argument, even if there are no naming conflicts. For example, <code>model("post").findAll(include="comment")</code> in a fictitious blog application would yield these column names: <code>id</code> , <code>title</code> , <code>authorId</code> , <code>body</code> , <code>createdAt</code> , <code>commentID</code> , <code>commentName</code> , <code>commentBody</code> , <code>commentCreatedAt</code> , <code>commentDeletedAt</code> . When this setting is set to <code>false</code> , the returned column list would look like this: <code>id</code> , <code>title</code> , <code>authorId</code> , <code>body</code> , <code>createdAt</code> , <code>commentID</code> , <code>name</code> , <code>commentBody</code> , <code>commentCreatedAt</code> , <code>deletedAt</code> .

Plugin Settings

There are several settings that make plugin development more convenient. We recommend only changing these settings in design or development modes so there aren't any deployment issues in production, testing, and maintenance modes. (At that point, your plugin should be properly packaged in a zip file.)

If you want to keep what's stored in a plugin's zip file from overwriting changes that you made in its expanded folder, set this in `config/design/settings.cfm` and/or `config/development/settings.cfm`:

```
<cfset set(overwritePlugins=false)>
```

See the chapter on Using and Creating Plugins for more information.

Name	Type	Default	Description
deletePluginDirectories	boolean	true	When set to <code>true</code> , Wheels will remove subdirectories within the <code>plugins</code> folder that do not contain corresponding plugin zip files. Set to <code>false</code> to add convenience to the process for developing your own plugins.
loadIncompatiblePlugins	boolean	true	Set to <code>false</code> to stop Wheels from loading plugins whose supported versions do not match your current version of Wheels.
overwritePlugins	boolean	true	When set to <code>true</code> , Wheels will overwrite plugin files based on their source zip files on application reload. Setting this to <code>false</code> makes plugin development easier because you don't need to keep unzipping your plugin files every time you make a change.

Miscellaneous Settings

How about situations that don't fit into those previous 5 categories? Well, they all fall right into this miscellaneous section.

Let's say you don't like the convention name for the soft delete feature of Wheels. Changing it is as easy as this:

```
<cfset set(softDeleteProperty#trashedAt)>
```

This will enable soft delete on all models that contains the `trashedAt` property instead of the Wheels default, `deletedAt`.

Full Listing of Miscellaneous Settings

Name	Type	Default	Description
assetQueryString	boolean	false in design and development environments, true in the others	Set to <code>true</code> to append a unique query string based on a time stamp to JavaScript, CSS, and image files included with the media view helpers. This helps force local browser caches to refresh when there is an update to your assets. This query string is updated when reloading your Wheels application. You can also hard code it by passing in a string.
assetPaths	struct	false	Pass <code>false</code> or a struct with up to 2 different keys to autopopulate the domains of your assets: <code>http</code> (required) and <code>https</code> . For example: { <code>http</code> ="asset0.domain1.com,asset2.domain1.com,asset3.domain1.com", <code>https</code> ="secure.domain1.com" }
ipExceptions	string	[empty string]	IP addresses that Wheels will ignore when the environment is set to maintenance. That way administrators can test the site while in maintenance mode, while the rest of users will see the message loaded in <code>events/onmaintenance.cfm</code> .
loadDefaultRoutes	boolean	true	Set to <code>false</code> to disable Wheels's default route patterns for <code>controller/action/key</code> , etc.
obfuscateUrls	boolean	false	Set to <code>true</code> to obfuscate primary keys in URLs.
reloadPassword	string	[empty string]	Password to require when reloading the Wheels application from the URL. Leave empty to require no password.

Wrapping It Up

There are literally hundreds of configurations options in Wheels for you to play around with. So go ahead and sink your teeth into Wheels configuration and defaults.

Directory Structure

Finding your way around a Wheels application.

After downloading and unzipping Wheels, here's the directory structure that you will see (plus signs indicate folders):

- + config
- + controller
- + events
- + files
- + images
- + javascripts
- + miscellaneous
- + model
- + plugins
- + stylesheets
- + view
- + wheels
- .htaccess
- Application.cfc
- index.cfm
- IsapiRewrite4.ini
- rewrite.cfm
- root.cfm
- web.config

Quick Summary

Your configuration settings will be done in the `config` directory.

Your application code will end up in four of the folders, namely `controllers`, `events`, `models`, and `views`.

Static media files should be placed in the `files`, `images`, `javascripts` and `stylesheets` folders.

Place anything that need to be executed outside of the framework in the `miscellaneous` folder. The framework does not get involved when executing `.cfm` files in this folder. (The empty `Application.cfc` takes care of that.) Also, no URL rewriting will be performed in this folder, so it's a good fit for placing CFCs that need to be accessed remotely via `<cfajaxproxy>` and Flash AMF binding, for example.

Place Wheels plugins in the `plugins` folder.

And the last directory? That's the framework itself. It exists in the `wheels` directory. Please go in there and have a look around. If you find anything you can improve or new features that you want to add, let us know!

Detailed Overview

Let's go through all of the files and directories now starting with the ones you'll spend most of your time in: the code directories.

controllers

This is where you create your controllers. You'll see two files in here already: `Controller.cfc` and `wheels.cfc`. You can place functions inside `Controller.cfc` to have that function shared between all the controllers you create. (This works because all your controllers will extend `Controller`.) `wheels.cfc` is an internal file used by Wheels.

models

This is where you create your model files (or classes if you prefer that term). Each model file you create should map to one table in the database.

The setup in this directory is similar to the one for controllers, to share methods you can place them in the existing `Model.cfc` file.

views

This is where you prepare the views for your users. As you work on your website, you will create one view directory for each controller.

events

If you want code executed when ColdFusion triggers an event, you can place it here (rather than directly in `Application.cfc`).

config

Make all your configuration changes here. You can set the environment, routes, and other settings here. You can also override settings by making changes in the individual settings files that you see in the subdirectories.

files

Any files that you intend to deliver to the user using the `sendFile()` function should be placed here. Even if you don't use that function to deliver files, this folder can still serve as file storage if you like.

images

This is a good place to put your images. It's not required to have them here, but all Wheels functions that involve images will, by convention, assume they are stored here.

javascripts

This is a good place to put your JavaScript files.

stylesheets

This is a good place to put your CSS files.

miscellaneous

Use this folder for code that you need to run completely outside of the framework. (There is an empty `Application.cfc` file in here, which will prevent Wheels from taking part in the execution.)

This is most useful if you're using Flash to connect directly to a CFC via AMF binding or if you're using `<cfajaxproxy>` in your views to bind directly to a CFC as well.

plugins

Place any plugins you have downloaded and want installed here.

wheels

This is the framework itself. When a new version of Wheels is released it is often enough to just drop it in here (unless there has been changes to the general folder structure).

.htaccess

This file is used by Apache, and you specifically need it for URL rewriting to work properly. If you're not using Apache, then you can safely delete it.

IsapiRewrite4.ini

If you use IIS 6 and want to use URL Rewriting, you'll need this file. Otherwise, you can safely delete it.

web.config

Same as `IsapiRewrite4.ini` but used for version 7 of IIS instead.

Application.cfc, index.cfm, root.cfm, and rewrite.cfm

These are all needed for the framework to run. No changes should be done to these files.

You can add more directories if you want to, of course. Just remember to include a blank `Application.cfc` in those directories. Otherwise, Wheels will try to get itself involved with those requests as well.

Switching Environments

Environments that match your development stages.

Wheels allows you to set up different *environments* that match stages in your development cycle. That way you can configure different values that match what services to call and how your app behaves based on where you are in your development.

By default, all new applications will start in the Design environment. The Design environment is the most convenient one to use as you start building your application because it does not cache any data. Therefore, if you make any changes to your database, for example, it will immediately be picked up by Wheels.

Other environment modes caches this information in order to speed up your application as much as possible. Making changes to the database in these environment modes will cause Wheels to throw an error. (Although that can be avoided with a `reload` call. More on that later.)

The fastest environment mode in terms of page load time is the Production mode. This is what you should set your application to run in before you launch your website.

The 5 Environment Modes

Besides the 2 environments mentioned above, there are 3 more. Let's go through them all one by one so you can see the differences between them and choose the most appropriate one given your current stage of development.

Design

- Shows friendly Wheels specific errors as well as regular ColdFusion errors on screen.
- Does not email you when an error is encountered.
- All caches are cleared before each request, except for settings defined in any of the files in the `config` folder.

Development

- Handles errors the same way as the Design mode.
- Caches controller and model initialization (the `init()` methods).
- Caches the database schema.
- Caches routes.
- Caches image information.

Production

- Caches everything that the Development mode caches.
- Activates all developer controlled caching (actions, pages, queries and partials).
- Shows your custom error page when an error is encountered.
- Shows your custom 404 page when a controller or action is not found.
- Sends an email to you when an error is encountered.

Testing

- Same caching settings as the Production mode but using the error handling of the Development mode. (Good for testing an application at its true speed while still getting errors reported on screen.)

Maintenance

- Shows your custom maintenance page unless the requesting IP address is in the exception list (set by calling `set(ipExceptions="127.0.0.1")` in `config/settings.cfm` or passed along in the URL as `except=127.0.0.1`).

This environment mode comes in handy when you want to briefly take your website offline to upload changes or modify databases on production servers.

How to Switch Modes

You change the current environment by modifying the `config/environment.cfm` file. After you've modified it, you need to either restart the ColdFusion service or issue a `reload` request. (See below for more info.)

The `reload` Request

Issuing a `reload` request is the easiest way to go from one environment to another. It's done by passing in `reload` as a parameter in the URL, like this:

```
■ http://www.mysite.com/?reload=true
```

This tells Wheels to reload the entire framework (it will also run your code in the `events/onapplicationstart.cfm` file), thus picking up any changes made in the `config/environment.cfm` file.

Lazy Reloading

There's also a shortcut for lazy developers who don't want to change this file at all. To use it, just issue the `reload` request like this instead:

```
■ http://www.mysite.com/?reload=testing
```

This will make Wheels skip your `config/environment.cfm` file and just use the URL value instead (`testing`, in this case).

Password-Protected Reloads

For added protection, you can set the `reloadPassword` variable in `config/settings.cfm`. When set, a `reload` request will only be honored when the correct password is also supplied, like this:

```
■ http://www.mysite.com/?reload=testing&password=myspass
```

Using and Creating Plugins

Extend `Wheels` functionality by using plugins or creating your own.

`Wheels` is a fairly lightweight framework, and we like to keep it that way. We won't be adding thousands of various features to `Wheels` just because a couple of developers find them "cool." ;)

Our intention is to only have functionality we consider "core" inside of `Wheels` itself and then encourage the use of *plugins* for everything else.

By using plugins created by the community or yourself, you're able to add brand new functionality to `Wheels` or completely change existing features. The possibilities are endless.

Plugins are also the recommended way to get new code accepted into `Wheels`. If you have written code that you think constitutes core functionality and should be added to `Wheels`, then write a plugin. After the community has used it for a while, it will be a simple task for us to integrate it into `Wheels` itself.

Installing and Uninstalling Plugins

This couldn't be any simpler. To install a plugin, just download the plugin's `zip` file and drop it in the `plugins` folder.

If you want to remove it later simply delete the `zip` file. (`Wheels` will clean up any leftover folders and files.)

Reloading `Wheels` is required when installing/uninstalling. (Issue a `reload=true` request.)

File Permissions on `plugins` Folder

You may need to change access permissions on your application's `plugins` folder so that `Wheels` can write the subfolders and files that it needs to run. If you get an error when testing out a plugin, you may need to loosen up the permission level.

Plugin Naming, Versioning, and Dependencies

When you download plugins, you will see that they are named something like this: `Scaffold-0.1.zip`. In this case, `0.1` is the version number of the `Scaffold` plugin. If you drop both `Scaffold-0.1.zip` and `Scaffold-0.2.zip` in the `plugins` folder, `Wheels` will use the one with the highest version number and ignore any others.

If you try to install a plugin that is not compatible with your installed version of `Wheels` or not compatible with a previously installed plugin (i.e., they try to add/override the same functions), `Wheels` will throw an error on application start.

If you install a plugin that depends on another plugin, you will get a warning message displayed in the debug area. This message will name the plugin that you'll need to download and install to make the originally installed plugin work correctly.

Available Plugins

To view all official plugins that are available for Wheels you can go to the Plugins Directory on our website.

While there, you can also download the PluginManager It lets you browse and install plugins directly from inside your Wheels application. It's quite handy!

Creating Your Own Plugins

To create a plugin named `MyPlugin`, you will need to create a `MyPlugin.cfc` and an `index.cfm` file. Then zip these together as `MyPlugin-x.x.zip`, where `x.x` is the version number of your plugin.

The only other requirement to make a plugin work is that `MyPlugin.cfc` must contain a method named `init`. This method must set a variable called `this.version`, specifying the Wheels version the plugin is meant to be used on (or several Wheels version numbers in a list) and then return itself.

Here's an example:

```
<cfcomponent output=false>
    <cffunction name=init>
        <cfset this.version = "1.0,1.1">
        <cfreturn this>
    </cffunction>
</cfcomponent>
```

The `index.cfm` file is the user interface for your plugin and will be viewable when clicking through to it from the debug area of your application.

Using a Plugin to Add or Alter Capabilities

A plugin can add brand new functions to Wheels or override existing ones. A plugin can also have a simple one-page user interface so that the users of the plugin can provide input, display content, etc.

To add or override a function, you simply add the function to `MyPlugin.cfc`, and Wheels will inject it into Wheels on application start.

Please note that all functions in your plugin need to be public (`access="public"`). If you have functions that should only be called from the plugin itself, we recommend starting the function name with the `$` character (this is how many internal Wheels functions are named as well) to avoid any naming collisions.

It is also important to note that although you can overwrite functions, they are still available for you to leverage with the use of `core.functionName()`.

Example: Overriding `timeAgoInWords()`

Let's say that we wanted Wheels's built-in function `timeAgoInWords()` to return the time followed by the string " (approximately)":

```
<cffunction name="timeAgoInWords" returntype="string" access="public" output="false">
  <cfreturn core.timeAgoInWords(argumentCollection=arguments) (@approximately)>
</cffunction>
```

Plugin Attributes

There are 3 attributes you can set on your plugin to customize its behavior. The first and most important one is the `mixin` attribute.

By default, the functions in your plugins will be injected into all Wheels objects (controller, model, etc.). This is usually not necessary, and to avoid this overhead, you can use the `mixin` attribute to specify exactly where the functions should be injected. If you have a function that should be available in a controller (or view) this is how it could look:

```
<cfcomponent output="false" mixin="controller">
```

The `mixin` attribute can be set either on the `cfcomponent` tag or on the individual `cffunction` tags.

The following values can be used for the `mixin` attribute: `application`, `global`, `none`, `controller`, `model`, `dispatch`, `microsoftsqlserver`, `mysql`, `oracle`, `postgresql`.

Another useful attribute is the `environment` attribute. Using this, you can tell Wheels that a plugin should only inject its functions in certain environment modes. Here's an example of that, taken from the Scaffold plugin, which doesn't need to inject any functions to Wheels at all.

```
<cfcomponent output="false" mixin="controller" environment="design,development">
```

Finally, there is a way to specify that your plugin needs another plugin installed in order to work. Here's an example of that:

```
<cfcomponent output="false" dependency="someOtherPlugin">
```

Making Plugin Development More Convenient with Wheels Settings

When your Wheels application first initializes, it will unzip and cache the zip files in the `plugins` folder. Each plugin then has its own expanded subfolder. If a subfolder exists but has no corresponding zip file, Wheels will delete the folder and its contents.

This is convenient when you're deploying plugins but can be annoying when you're developing your own plugins. By default, every time you make a change to your plugin, you need to rezip your plugin files and reload the Wheels application by adding `?reload=true` to the URL.

Disabling Plugin Overwriting

To force Wheels to skip the unzipping process, set the `overwritePlugins` setting to `false` for your `design` and/or

development environments.

```
<!-- In config/design/settings.cfm -->  
<cfset set(overwritePlugins=false)
```

With this setting, you'll be able to reload your application without worrying about your file being overwritten by the contents of the corresponding zip file.

Disabling Plugin Folder Deletion

To force Wheels to skip the folder deletion process, set the `deletePluginDirectories` setting to `false` for your design and/or development environments.

```
<!-- In config/design/settings.cfm -->  
<cfset set(deletePluginDirectories=false)
```

With this setting, you can now develop new plugins in your application without worrying about having a corresponding zip file in place.

See the chapter on Configurations and Defaults for more details about changing Wheels settings.

Stand-Alone Plugins

If your plugin is completely stand-alone, you can call it from its view page using just the name of the plugin. This works because Wheels has created a pointer to the plugin object residing in the `application` scope. One example of a stand-alone plugin is the `PluginManager`. If you check out its view code, you will see that it calls itself like this:

```
<cfset pluginManager.installPlugin(URL.plugin)
```

Now Go Build Some Plugins!

Armed with this knowledge about plugins, you can now go and add that feature you've always wanted or change that behavior you've always hated. We've stripped you of any right to blame us for your discontents. :)

Request Handling

How Wheels handles an incoming request.

Wheels is quite simple when it comes to figuring out how incoming requests map to code in your application. Let's look at a URL for an e-commerce website and dissect it a little.

But before we do that, a quick introduction to URLs in Wheels is in order.

A Wheels URL

URLs in Wheels generally look something like this:

```
■ http://localhost/index.cfm/shop/products
```

It's also possible that the URLs will look like this in your application:

```
■ http://localhost/index.cfm?controller=shop&action=products
```

This happens when your web server does not support the `cgi.path_info` variable.

Regardless of your specific setup, Wheels will try to figure out how to handle the URLs. If Wheels fails to do this properly (i.e. you *know* that your web server supports `cgi.path_info`, but Wheels insists on creating the URLs with the query string format), you can override it by setting `URLRewriting` in `config/settings.cfm` to either `On`, `Partial`, or `Off`. The line of code should look something like this:

```
■ <cfset set(URLRewriting="Partial")>
```

"Partial URL Rewriting" is what we call that first URL up there with the `/index.cfm/` format.

In the example URLs used above, `shop` is the name of the controller to call, and `products` is the name of the action to call on that controller.

Model-View-Controller Explained

Unless you're familiar with the Model-View-Controller pattern, you're probably wondering what controllers and actions are.

Put very simply, a *controller* takes an incoming request and, based on the parameters in the URL, decides what (if any) data to get from the model (which in most cases means your database), and decides which view (which in most cases means a CFML file producing HTML output) to display to the user.

An *action* is an entire process of executing code in the controller, including a view file and rendering the result to the browser. As you will see in the example below, an action usually maps directly to one specific function with the same name in the controller file.

Creating URLs

Mapping an incoming URL to code is only one side of the equation. You will also need a way to create these URLs. This is done through a variety of different functions like `linkTo()` (for creating links), `startFormTag()` (for creating forms), and `redirectTo()` (for redirecting users), to name a few.

Internally, all of these functions use the same code to create the URL, namely the `URLFor()` function. The `URLFor()` function accepts a `controller` and an `action` argument, which are what you will use most of the time. It has a lot of other arguments and does some neat stuff (like defaulting to the current controller when you don't specifically pass one in). So check out the documentation for the `URLFor()` function for all the details.

By the way, by using URL rewriting in Apache or IIS, you can completely get rid of the `index.cfm` part of the URL so that `http://localhost/index.cfm/shop/products` becomes `http://localhost/shop/products`. You can read more about this in the URL Rewriting chapter.

For the remainder of this chapter, we'll type out the URLs in this shorter and prettier way.

A Wheels Page

Let's look a little closer at what happens when Wheels receives this example incoming request.

First, it will create an instance of the `shop` controller (`controllers/Shop.cfc`) and call the function inside it named `products`.

Let's show how the code for the `products` function could look to make it more clear what goes on:

```
<cfcomponent extends="Controller">
    <cffunction name="products">
        <cfset renderPage(controller="shop", action="products")>
    </cffunction>
</cfcomponent>
```

The only thing this does is specify the view page to render using the `renderPage()` function.

The `renderPage()` function is available to you because the `shop` controller extends the main `Wheels Controller` component. Don't forget to include that `extends` attribute in your `<cfcomponent>` tags as you build your controllers!

So, how does `renderPage()` work? Well, it accepts the arguments `controller` and `action` (among others), and, based on these, it will try to include a view file. In our case, the view file is stored at `views/shop/products.cfm`.

You can read the chapter about Rendering Content for more information about the `renderPage()` function.

It's important to note that the `renderPage()` function does not cause any controller actions or functions to be executed. It just specifies what view files to get content from. Keep this in mind going forward because it's a common assumption that it does. (Especially when you want to include the view page for another action, it's easy to jump to the incorrect conclusion that the code for that action would also get executed.)

Wheels Conventions

Because Wheels favors convention over configuration, we can remove a lot of the code in the example above, and it will still work because Wheels will just guess what your intention is. Let's have a quick look at exactly what code can be removed and why.

The first thing Wheels assumes is that if you call `renderPage()` without arguments, you want to include the view page for the **current** controller and action.

Therefore, the code above can be changed to:

```
<cfcomponent extends="Controller">
    <cffunction name="products">
        <cfset renderPage()>
    </cffunction>
</cfcomponent>
```

... and it will still work just fine.

Does Wheels assume anything else? Sure it does. You can actually remove the entire `renderPage()` call because Wheels will assume that you always want to call a view page when the processing in the controller is done. Wheels will call it for you behind the scenes.

That leaves you with this code:

```
<cfcomponent extends="Controller">
    <cffunction name="products">
    </cffunction>
</cfcomponent>
```

That looks rather silly, a `products` function with no code whatsoever. What do you think will happen if you just remove that entire function, leaving you with this code?

```
<cfcomponent extends="Controller">
</cfcomponent>
```

...If you guessed that Wheels will just assume you don't need any code for the `products` action and just want the view rendered directly, then you are correct.

In fact, if you have a completely blank controller like the one above, you can delete it from the file system altogether!

This is quite useful when you're just adding simple pages to a website and you don't need the controller and model to be involved at all. For example, you can create a file named `about.cfm` in the `views/home` folder and access it at `http://localhost/home/about` without having to create a specific controller and/or action for it.

This also highlights the fact that Wheels is a very easy framework to get started in because you can basically program just as you

normally would be by creating simple pages like this and then gradually "Wheelsifying" your code as you learn the framework.

The `params` Struct

Besides making sure the correct code is executed, `Wheels` also does something else to simplify request handling for you. It combines the `url` and `form` scopes into one. This is something that most CFML frameworks do as well. In `Wheels`, it is done in the `params` struct.

The `params` struct is available to you in the controller and view files but not in the model files. (It's considered a best practice to not mix your request handling with your business logic.) Besides the `form` and `url` scope variables, the `params` struct also contains the current controller and action name for easy reference.

If the same variable exists in both the `url` and `form` scopes, the value in the `form` scope will take precedence.

To make this concept easier to grasp, imagine a login form on your website that submits to `http://localhost/account/login?sendTo=dashboard` with the variables `username` and `password` present in the form. Your `params` struct would look like this:

Name	Value
<code>params.controller</code>	<code>account</code>
<code>params.action</code>	<code>login</code>
<code>params.sendTo</code>	<code>dashboard</code>
<code>params.username</code>	<code>joe</code>
<code>params.password</code>	<code>1234</code>

Now instead of accessing the variables as `url.sendTo`, `form.username`, etc., you can just use the `params` struct for all of them instead.

This concept becomes even more useful once we start getting into creating forms specifically meant for accessing object properties. But let's save the details of all that for the `Form Helpers` and `Showing Errors` chapter.

Routing

For more advanced URL-to-code mappings, you can use a concept called *routing*. It allows for you to fully customize every URL in your application. You can read more about this in the chapter called `Using Routes`.

Rendering Content

Showing content to the user.

A big part of a controller's task is to respond to the user. In Wheels you can respond to the user in three different ways:

- Displaying content
- Redirecting to another URL
- Sending a file

You can only respond once per request. If you do not explicitly call any of the response functions (`renderPage()`, `sendFile()` etc) then Wheels will assume that you want to show the view for the current controller and action and do it for you.

This chapter covers the first method listed above—displaying content. The chapters about Redirecting Users and Sending Files cover the other two response methods.

Rendering a Page

This is the most common way of responding to the user. It's done with the `renderPage()` function, but most often you probably won't call it yourself and instead let Wheels do it for you.

Sometimes you will want to call it though and specify to show a view page for a controller/action other than the current one. One common technique for handling a form submission, for example, is to show the view page for the controller/action that contains the form (as opposed to the one that just handles the form submission and redirects the user afterwards). When doing this, it's very important to keep in mind that `renderPage()` will **not run the code** for the controller's action—all it does is process the view page for it.

You can also call `renderPage()` explicitly if you wish to cache the response or use a different layout than the default one.

If the `controller` and `action` arguments do not give you enough flexibility, you can use the `template` argument that is available for `renderPage()`.

Refer to the Pages chapter for more details about rendering content. More specifically, that chapter describes where to place those files and what goes in them.

Rendering a Partial

This is done with the `renderPartial()` function. It's most often used with AJAX requests that are meant to update only parts of a page.

Rendering Nothing at All

Sometimes you don't need to return anything at all to the browser. Perhaps you've made an AJAX request that does not require a response or executed a scheduled task that no end user sees the results of. In these cases you can use the `renderNothing()` function to tell Wheels to just render an empty page to the browser.

Rendering Text

This is done with the `renderText()` function. It just returns the text you specify. In reality it is rarely used but could be useful as a response to AJAX requests sometimes.

Rendering to a Variable

Normally when you call any of the rendering functions, the result is stored inside an internal Wheels variable. This value is then outputted to the browser at the end of the request.

Sometimes you may want to do some additional processing on the rendering result before outputting it though. This is where the `returnAs` argument comes in handy. It's available on both `renderPage()` and `renderPartial()`. Setting `returnAs` to `string` will return the result to you instead of placing it in the internal Wheels variable.

Caching the Response

Two of the functions listed above are capable of caching the content for you; `renderPage()` and `renderPartial()`. Just pass in `cache=true` (to use the default cache time set in `config/settings.cfm`) or `cache=x` where `x` is the number of minutes you want to cache the content for. Keep in mind that this caching respects the global setting set for it in your configuration files so normally no pages will be cached when in Design or Development mode.

We cover caching in greater detail in the Caching chapter.

Using a Layout

The `renderPage()` function accepts an argument named `layout`. Using this you can wrap your content with common header/footer style code. This is such an important concept though so we'll cover all the details of it in the chapter called Using Layouts.

Redirecting Users

Use redirection to keep your application user friendly.

When a user submits a form, you do **not** want to show any content on the page that handles the form submission! Why? Because if you do, and the user hits refresh in their browser, the form handling code could be triggered again, possibly causing duplicate entries in your database, multiple emails being sent, etc.

Remember to Redirect

To avoid the above problem, it is recommended to always redirect the user after submitting a form. In Wheels this is done with the `redirectTo()` function. It is basically a wrapper around the `cflocation` tag in CFML.

Being that `redirectTo()` is a Wheels function, it can accept the `route`, `controller`, `action`, and `key` arguments so that you can easily redirect to other actions in your application.

Three Ways to Redirect

Let's look at the three ways you can redirect in Wheels.

1. Redirecting to Another Action

You can redirect the user to another action in your application simply by passing in the `controller`, `action`, and `key` arguments. You can also pass in any other arguments that are accepted by the `URLFor()` function, like `host`, `params`, etc. (The `URLFor()` function is what Wheels uses internally to produce the URL to redirect to.)

2. Redirection Using Routes

If you have configured any routes in `config/routes.cfm`, you can use them when redirecting as well. Just pass in the route's name to the `route` argument together with any additional arguments needed for the route in question. You can read more about routing in the Using Routes chapter.

3. Redirecting to the Referring URL

It's very common that all you want to do when a user submits a form is send them back to where they came from. (Think of a user posting a comment on a blog post and then being redirected back to view the post with their new comment visible as well.) For this, we have the `back` argument. Simply pass in `back=true` to `redirectTo()`, and the user will be redirected back to the page they came from.

Handling an Invalid Referrer

The referring URL is retrieved from the `cgi.http_referer` value. If this value is blank or comes from a different domain than the current one, Wheels will redirect the visitor to the root of your website instead.

If you want to specify exactly where to send the visitor when the referring domain is blank/foreign, you can pass in the normal `URLFor()` arguments like `route`, `controller`, `action`, etc. These will be used only when Wheels can't redirect to the referrer because it's invalid.

Appending Params

Sometimes it's useful to be able to send the visitor back to the same URL they came from but with extra parameters added to it. You can do this by using the `params` argument. Note that Wheels will append to the URL and not replace it in this case.

`addToken` and `statusCode` Arguments

Sending Files

Use Wheels to send files to your users securely and with better control of the user experience.

Sending files?! Is that really a necessary feature of Wheels? Can't I just place the file on my web server and link to it? You are correct, there is absolutely no need to use Wheels to send files. Your web server will do a fine job of sending out files to your users.

However, if you want a little more control over the way the user's browser handles the download or be able to secure access to your files then you might find the `sendFile()` function useful.

Sending Files With the `sendFile()` Function

The convention in Wheels is to place all files you want users to be able to download in the `files` folder.

Assuming you've placed a file named `wheels_tutorial_20081028_J657D6HX.pdf` in that folder, here is a quick example of how you can deliver that file to the user. Let's start with creating a link to the action that will handle the sending of the file first.

```
#linkTo(text="Download Tutorial", action="sendTutorial")#
```

Now let's send the file to the user in the `sendTutorial` controller action. Just like the rendering functions in Wheels, the `sendFile()` function should be placed as the very last thing you do in your action code.

In this case, that's the only thing we are doing, but perhaps you want to track how many times the file is being downloaded, for example. In that case, the tracking code needs to be placed *before* the `sendFile()` function.

Also, remember that the `sendFile()` function replaces any rendering. You cannot send a file *and* render a page. (This will be quite obvious once you try this code because you'll see that the browser will give you a dialog box, and you won't actually leave the page that you're viewing at the time.)

Here's the `sendTutorial` action:

```
<cffunction name="sendTutorial">
  <cfset sendFile(file="wheels_tutorial_20081028_J657D6HX.pdf")>
</cffunction>
```

That's one ugly file name though, eh? Let's present it to the user in a nicer way by suggesting a different name to the browser:

```
<cffunction name="sendTutorial">
  <cfset sendFile(file="wheels_tutorial_20081028_J657D6HX.pdf", name="Tutorial.pdf")>
</cffunction>
```

Much better! :)

By default, the `sendFile()` function will try and force a download dialog box to be shown to the user. The purpose of this is to

make it easy for the user to save the file to their computer. If you want the file to be shown inside the browser instead (when possible as decided by the browser in question), you can set the `disposition` argument to `inline`.

Here's an example:

```
<cffunction name="sendTutorial">
  <cfset sendFile(file="wheels_tutorial_20081028_J657D6HX.pdf",disposition="inline")>
</cffunction>
```

You can also specify what HTTP content type to use when delivering the file by using the `type` argument. Please refer to the API for the `sendFile()` function for complete details.

Securing Access to Files

Perhaps the main reason to use the `sendFile()` function is that it gives you an easy way to secure access to your files. Maybe the tutorial file used in the above example is something the user paid for, and you only want for that user to be able to download it (and no one else). To accomplish this, you can just add some code to authenticate the user right before the `sendFile()` call in your action.

However, there is a security flaw here. Can you figure out what it is?

You may have guessed that the `files` folder is placed in your web root, so anyone can download files from it by typing `http://www.domain.com/files/wheels_tutorial_20081028_J657D6HX.pdf` in their browser. Although users would need to guess the file names to be able to access the files, we would still need something more robust as far as security goes.

There are two solutions to this.

The easiest one is to just lock down access to the folder using your web server. Wheels won't be affected by it since it gets the file from the file system.

If that is not an option, the other option is simply to move the `files` folder out of the web root, thus making it inaccessible. If you move the folder, you'll need to accommodate for this in your code by changing your `sendFile()` calls to specify the path as well, like this:

```
<cffunction name="sendTutorial">
  <cfset sendFile(file="../../tutorials/wheels_tutorial_20081028_J657D6HX.pdf")>
</cffunction>
```

This assumes you've moved the folder two levels up in your file system and into a folder named `tutorials`.

Don't Open Any Holes with URL Parameters

A final note of warning: Be careful to not allow just any parameters from the URL to get passed through to the `sendFile()` because then a user would be able to download any file from your server by playing around with the URL. Be wary of how you're using the `params` struct in this context!

Sending Email

Use Wheels to simplify the task of setting up automated emails in your application.

Sending emails in Wheels is done from your controller files with the `sendEmail()` function. It's basically a wrapper around `cfmail`, but it has some smart functionality and a more Wheels-like approach in general.

Getting this to work in Wheels can be broken down in 3 steps. We'll walk you through them.

Establishing Mail Server and Account Information

We recommend using Wheels ability to set global defaults for `sendEmail()` so that you don't have to specify more arguments than necessary in your controller files. Because it's likely that you will use the same mail server across your application, this makes it worthwhile to set a global default for it.

This setting should be done in the `config/settings.cfm` file and can look something like this:

```
<cfset set(
  functionName="sendEmail",
  server=#yourServer",
  username=#yourUsername",
  password=#yourPassword"
)>
```

By specifying these values here, these arguments can be omitted from all `sendEmail()` function calls, thus providing cleaner, less cluttered code.

But you are not limited to setting only these 3 variables. In fact, you can set a global default for any optional argument to `sendEmail()` and since it accepts the same arguments that `cfmail` does. That's quite a few.

Create an Email Template

An email template is required for `sendEmail()` to work and forms the basis for the mail message content. Think of an email template as the content of your email.

Templates may be stored anywhere within the `/views/` folder, but we recommend a structured, logical approach. If different controllers utilize `sendEmail()` and each require a unique template, place each email template within the `views/controllername` folder structure.

Consider this example scenario:

Controller:	Membership
Email Template:	<code>/views/membership/myemailtemplate.cfm</code>

Multiple templates may be stored within this directory should there be a need.

The content of the template is simple: simply output the content and any expected variables.

Here's an example for `myemailtemplate.cfm`, which will contain HTML content.

```
<cfoutput>
<p>Hi #recipientName#,/p>
<p>
  We wanted to take a moment to thank you for joining our service
  and to confirm your start date.
</p>
<p>
  Our records indicate that your membership will begin on
  <strong>#DateFormat(startDate, "dddd, mmmm, d, yyyy")</strong>
</p>
</cfoutput>
```

Sending the Email

As we've said before, `sendEmail()` accepts all attribute of CFML's `cfmail` tag as arguments. But it also accepts any variables that you need to pass to the email template itself.

Consider the following example:

```
<cfset member = model("member").findByKey(newMember.id)
<cfset
  sendEmail(
    from=#service@yourwebsite.com"
    to=member.email,
    template="myemailtemplate,"
    subject="Thank You for Becoming a Member"
    recipientName=member.name,
    startDate=member.startDate
  )
>
```

Here we are sending an email by including the `myemailtemplate` template and passing values for `recipientName` and `startDate` to it.

Note that the `template` argument should be the path to the view's folder name and template file name without the extension. If the template is in the current controller, then you don't need to specify a folder path to the template file. In that case, just be sure to store the template file in the folder with the rest of the views for that controller.

The logic for which template file to include follows the same logic as the `template` argument to `renderPage()`.

Did you notice that we did not have to specify the `type` attribute of `cfmail`? Wheels is smart enough to figure out that you want to send as HTML since you have tags in the email body. (You can override this behavior if necessary though by passing in the `type` argument.)

Multipart Emails

The intelligence doesn't end there though. You can have Wheels send a multipart email by passing in a list of templates to the `templates` argument (notice the plural), and Wheels will automatically figure out which one is text and which one is HTML.

It is also important to note that the `template` argument should be the path to the view's folder name and template file name without the extension. If the template is in the current controller, then you don't need to specify a folder path to the template file. In that case, just be sure to store the template file in the folder with the rest of the views for that controller.

Like the `template` argument, the logic for which file to include follows the same logic as the `template` argument to `renderPage()`.

Attaching Files

You can attach files to your emails as well by using the `file` argument (or `files` argument if you want multiple attachments). Simply pass in the name of a file that exists in the `files` folder (or a subfolder of it) of your application.

Using Email Layouts

Much like the layouts outlined in the Using Layouts chapter, you can also create layouts for your emails.

A layout should be used just as the name implies: for layout and stylistic aspects of the email body. Based on the example given above, let's assume that the same email content needs to be sent twice.

1. Message is sent to a new member with a stylized header and footer.
2. A copy of message is sent to an admin at your company with a generic header and footer.

Best practice is that variables (such as `recipientName` and `startDate`, in the example above) be placed as outputs in the template file.

In this case, the 2 calls to `sendEmail()` would be nearly identical, with the exception of the `layout` attribute.

```
<!-- Get new member -->
<cfset member = model("member").findByKey(params.key)>

<!-- Customer email with customized header/footer -->
<cfset
    sendEmail(
        from="service@yourwebsite.com"
        template="myemailtemplate"
        layout="customer",
        to=member.email,
        subject="Thank You for Becoming a Member"
        recipientName=member.name,
        startDate=member.startDate
    )
>

<!-- Plain text message with "admin" layout -->
<cfset
```

```
sendEmail(  
    from=#service@yourwebsite.com"  
    template="myemailtemplate,"  
    layout="admin",  
    to=#admin@domain.com"  
    subject="Membership Verification: #member.name#"  
    recipientName=member.name,  
    startDate=member.startDate  
)  
>
```

Multipart Email Layouts

Wheels also lets you set up layouts for the HTML and plain text parts in a multipart email.

If we set up generic email layouts at `views/plainemaillayout.cfm` and `views/htmlemaillayout.cfm`, we would call `sendEmail()` like so:

```
<!--Multipart customer email -->  
<cfset  
    sendEmail(  
        from=#service@yourwebsite.com"  
        templates="/myemailtemplateplain,/myemailtemplatehtml"  
        layouts="customerPlain,customerHtml"  
        to=member.email,  
        subject="Thank You for Becoming a Member"  
        recipientName=member.name,  
        startDate=member.startDate  
    )  
>
```

For both the `templates` and `layouts` arguments (again, notice the plurals), we provide a list of view files to use. Wheels will figure out which of the templates and layouts are the HTML versions and separate out the MIME parts for you automatically.

Go Send Some Emails

Now you're all set to send emails the Wheels way. Just don't be a spammer, please!

Responding with Multiple Formats

Wheels controllers provide some powerful mechanisms for responding to requests for content in XML, JSON, and other formats. Build an API with ease with these functions.

If you've ever needed to create an XML or JSON API for your Wheels application, you may have needed to go down the path of creating a separate controller or separate actions for the new format. This introduces the need to duplicate model calls or even break them out into a super long list of before filters. With this, your controllers can get pretty hairy pretty fast.

Using a few Wheels functions, you can easily respond to requests for HTML, XML, JSON, and PDF formats without adding unnecessary bloat to your controllers.

Requesting Different Formats

With Wheels Provides functionality in place, you can request different formats using the following methods:

1. URL Variable
2. URL Extension
3. Request Header

Which formats you can request is determined by what you configure in the controller. See the section below on *Responding to Different Formats in the Controller* for more details.

1. URL Variable

Wheels will accept a URL variable called `format`. If you wanted to request the XML version of an action, for example, your URL call would look something like this:

```
■ http://www.example.com/products?format=xml
```

The same would go for JSON:

```
■ http://www.example.com/products?format=json
```

2. URL Extension

Perhaps a cleaner way is to request the format as a "file" extension. Here are the XML and JSON examples, respectively:

```
■ http://www.example.com/products.xml  
■ http://www.example.com/products.json
```

This works similarly to the *URL variable* approach mentioned above in that there will now be a key in the `params` struct set to the `format` requested. With the XML example, there will be a variable at `params.format` with a value of `xml`.

3. Request Header

If you are calling the Wheels application as a web service, you can also request a given format via the HTTP `Accept` header.

If you are consuming the service with another Wheels application, your `<cfhttp>` call would look something like this:

```
<cfhttp url=#http://www.example.com/products"
  <cfhttpparam type="header" name="Accept" value="application/octet-stream"
</cfhttp>
```

In this example, we are sending an `Accept` header with the value for the `xml` format.

Here is a list of values that you can grab from `mimeTypes()` with Wheels out of the box.

1. `html`
2. `xml`
3. `json`
4. `csv`
5. `pdf`
6. `xls`

You can use `addFormat()` to set more types to the appropriate MIME type for reference. For example, we could set a Microsoft Word MIME type in `config/settings.cfm` like so:

```
<cfset addFormat(extension="doc", mimeType="application/msword">
```

Responding to Different Formats in the Controller

The fastest way to get going with creating your new API and formats is to call `provides()` from within your controller's `init()` method.

Take a look at this example:

```
<cfcomponent extends="Controller">
  <cffunction name="init">
    <cfset provides("html, json, xml">
  </cffunction>

  <cffunction name="index">
    <cfset products = model("product").findAll(order="title">
    <cfset renderWith(products)>
  </cffunction>
```

```
■ </cfcomponent>
```

By calling the `provides()` function in `init()`, you are instructing the Wheels controller to be ready to provide content in a number of formats. Possible choices to add to the list are `html` (which runs by default), `xml`, `json`, `csv`, `pdf`, and `xls`.

This is coupled with a call to `renderWith()` in the following actions. In the example above, we are setting a query result of `products` and passing it to `renderWith()`. By passing our data to this function, Wheels gives us the ability to respond to requests for different formats, and it even gives us the option to just let Wheels handle the generation of certain formats automatically.

When Wheels handles this response, it will set the appropriate MIME type in the `Content-Type` HTTP header as well.

Providing the HTML Format

Responding to requests for the HTML version is the same as you're already used to with Rendering Content. `renderWith()` will accept the same arguments as `renderPage()`, and you create just a view template in the `views` folder like normal.

Automatic Generation of XML and JSON Formats

If the requested format is `xml` or `json`, the `renderWith()` function will automatically transform the data that you provide it. If you're fine with what the function produces, then you're done!

Providing Your Own Custom Responses

If you need to provide content for another type than `xml` or `json`, or if you need to customize what your Wheels application generates, you have that option.

In your controller's corresponding folder in `views`, all you need to do is implement a view file like so:

Type	Example
<code>html</code>	<code>views/products/index.cfm</code>
<code>xml</code>	<code>views/products/index.xml.cfm</code>
<code>json</code>	<code>views/products/index.json.cfm</code>
<code>csv</code>	<code>views/products/index.csv.cfm</code>
<code>pdf</code>	<code>views/products/index.pdf.cfm</code>
<code>xls</code>	<code>views/products/index.xls.cfm</code>

If you need to implement your own XML- or JSON-based output, the presence of your new custom view file will override the automatic generation that Wheels normally performs.

Example: PDF Generation

If you need to provide a PDF version of the product catalog, the view file at `views/products/index.pdf.cfm` may look something like this:

```
■ <cfdocument format="pdf">
```

```
<h1>Products</h1>
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Description</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    #includePartial(products)#
  </tbody>
</table>
</cfdocument>
```

Using the Flash

Using the Flash to pass data from one request to the next.

The Flash is actually a very simple concept. And no, it has nothing to do with Adobe's Flash Player.

The Flash is just a struct in the `session` or `cookie` scope with some added functionality. It is cleared at the end of the next page that the user views. This means that it's a good fit for storing messages or variables temporarily from one request to the next.

By the way, the name "Flash" comes from Ruby on Rails, like so many other cool things in Wheels.

An Example of Using the Flash

The code below is commonly used in Wheels applications to store a message about an error in the Flash and then redirect to another URL, which then displays the message in its view page.

The following example shows how code dealing with the Flash can look in an action that handles a form submission.

```
<cfunction name="update">
  <cfif user.update(params.user)>
    <cfset flashInsert(success="The user was updated successfully")>
    <cfset redirectTo(action="edit")>
  <cfelse>
    <cfset renderPage(action="edit")>
  </cfif>
</cfunction>
```

Here's an example of how we then display the Flash message we just set in the view page for the `edit` action. Please note that this is done on the **next** request since we performed a redirect after setting the Flash.

```
<cfoutput>
<p class="success-message"
  #flash("success")#
</p>
</cfoutput>
```

As you can see above, you use the `flashInsert()` function with a named argument when you want to store data in the Flash and the `flash()` function when you want to display the data in a view.

The key chosen above is `success`, but it could have been anything that we wanted. Just like with a normal struct, the naming of the keys is your job.

As an example, you may choose to use one key for storing messages after the user made an error, called `error`, and another one for storing messages after a successful user operation, called `success`.

Shortcut for Setting the Flash and Redirecting

The more you work with Wheels and the Flash, the more that you're going to find that you keep repeating that `flashInsert()/redirectTo()` combo all the time. Wheels has a solution for that within the `redirectTo()` function itself:

```
()
```

That piece of code does exactly the same thing as the example shown previously in this chapter. The Wheels `redirectTo()` function sees the `success` argument coming in and knows that it's not part of its own declared arguments. Therefore, you (the developer) must intend for it to be stored in the Flash, so Wheels goes ahead and calls `flashInsert(success="The user was updated successfully.")()` for you behind the scenes.

Prepend with `flash` for Argument Names that Collide with `redirectTo()`'s

So what if you want to redirect to the `edit` action and set a key in the Flash named `action` as well? Simply prepend the key with `flash` to tell Wheels to avoid the argument naming collision.

```
<cfset redirectTo(action=edit", flashAction="The user was updated successfully">
```

We don't recommend naming the keys in your Flash `action`, but these naming collisions can potentially happen when you want to redirect to a route that takes custom arguments, so remember this workaround.

More Flashy Functions

Besides `flash()` and `flashInsert()` that are used to read from/insert to the Flash, there are a few other functions worth mentioning.

`flashCount()` is used to count how many key/value pairs there are in the Flash.

`flashClear()` and `flashDelete()` do exactly the same as their counterparts in the struct world, `StructClear` and `StructDelete`—they clear the entire Flash and delete a specific key/value from it, respectively.

`flashKeyExists()` is used to check if a specific key exists. So it would make sense to make use of that function in the code listed above to avoid outputting an empty `<p>` tag on requests where the Flash is empty. (`flash()` will return an empty string when the specified key does not exist.)

Check out the Controller Request Functions page for a listing of all the functions that deal with the Flash.

Wholesale Flash Handling with `flashMessages()`

Throw the `flashIsEmpty()` function into the mix, and you might find yourself writing code across your Wheels projects that looks something like this:

```
<cfoutput>  
<cfif NOT flashIsEmpty(>
```

```

<div id=#flash-messages#
  <cfif flashKeyExists#(error)#>
    <p class=#errorMessage#
      #flash("error")#
    </p>
  </cfif>
  <cfif flashKeyExists#(success)#>
    <p class=#successMessage#
      #flash("success")#
    </p>
  </cfif>
</div>
</cfif>
</cfoutput>

```

All of that above code can be replaced with a single call to the `flashMessages()` function:

```

<cfoutput>
#flashMessages()#
</cfoutput>

```

Whenever any value is inserted into the Flash, `flashMessages()` will display it similarly to the complex example above, with `class` attributes set similarly (`errorMessage` for the `error` key and `successMessage` for the `success` key).

You can also use `flashMessage()`'s `key/keys` argument to limit its reach to a list of given keys. Let's say that we only want our layout to show messages for the `alert` key but not for the `error` or `success` keys (or any other for that matter). We would write our call like so:

```

<cfoutput>
#flashMessages(key="alert")#
</cfoutput>

```

Just keep in mind that this approach isn't as flexible, so if you need to customize the markup of the messages beyond `flashMessages()`'s capabilities, you should revert back to using `flashIsEmpty()`, `flash()`, and other related functions manually.

Flash Storage Options

Earlier, we mentioned that the data for the Flash is stored in either the `cookie` or the `session` scope. You can find out where Wheels stores the Flash data in your application by outputting `get("flashStorage")`. If you have session management enabled in your application, Wheels will default to storing the Flash in the `session` scope. Otherwise, it will store it in a cookie on the user's computer.

You can override this setting in the same way that you override other Wheels settings by running the `set()` function like this:

```

<!-- In `config/settings.cfm` or another `settings.cfm` file within the
`config` subfolders -->
<cfset set(flashStorage=#session#)>

```

Note: Before you set Wheels to use the `session` scope, you need to make sure that session management is enabled. To enable it, all you need to do is add `this.SessionManagement = true` to the `config/app.cfm` file.

Choosing a Storage Method

So what storage option should you choose? Well, to be honest, it doesn't really matter that much, so we recommend you just go with the default setting. If you're a control freak and always want to use the optimal setting, here are some considerations.

- Although the Flash data is deserialized before stored in a cookie (making it possible to store complex values), you need to remember that a cookie is not the best place to store data that requires a lot of space.
- If you run multiple ColdFusion servers in a clustered environment and use session-based Flash storage, users might experience a loss of their Flash variables as their request gets passed to other servers.
- Using cookies is, generally speaking, less secure than using the `session` scope. Users could open their cookie file up and manually change its value. Sessions are stored on the server, out of users' reach.

Filters

Stop repeating yourself with the use of before and after filters.

If you find the need to run a piece of code before or after several controller actions, then you can use *filters* to accomplish this without needing to explicitly call the code inside each action in question.

This is similar to using the `OnRequestStart` / `OnRequestEnd` functions in CFML's `Application.cfc` file, with the difference being that filters tie in better with your Wheels controller setup.

An Example: Authenticating Users

One common thing you might find yourself doing is authenticating users before allowing them to see your content. Let's use this scenario to show how to use filters properly.

You might start out with something like this:

```
<cfcomponent extends="Controller">
    <cffunction name="secretStuff">
        <cfif NOT StructKeyExists(session,"userId")>
            <cfabort>
        </cfif>
    </cffunction>

    <cffunction name="evenMoreSecretStuff">
        <cfif NOT StructKeyExists(session,"userId")>
            <cfabort>
        </cfif>
    </cffunction>
</cfcomponent>
```

Sure, that works. But you're already starting to repeat yourself in the code. What if the logic of your application grows bigger? It could end up looking like this:

```
<cfcomponent extends="Controller">
    <cffunction name="secretStuff">
        <cfif cgi.remote_addr Does Not Contain 12.55">
            <cfset flashInsert(alert="Sorry, we're not open in that area")>
            <cfset redirectTo(action="sorry")>
        <elseif NOT StructKeyExists(session,"userId")>
            <cfset flashInsert(alert="Please login first.")>
            <cfset redirectTo(action="login")>
        </cfif>
    </cffunction>

    <cffunction name="evenMoreSecretStuff">
```

```

        <cfif cgi.remote_addr Does Not Contain 12.55">
            <cfset flashInsert(msg="Sorry, we're not open in that area")>
            <cfset redirectTo(action="sorry")>
        <cfelseif NOT StructKeyExists(session,"userId")>
            <cfset flashInsert(msg="Please login first.")>
            <cfset redirectTo(action="login")>
        </cfif>
    </cffunction>
</cfcomponent>

```

Ouch! You're now setting yourself up for a maintenance nightmare when you need to update that IP range, the messages given to the user, etc. One day, you are bound to miss updating it in one of the places.

As the smart coder that you are, you re-factor this to another method so your code ends up like this:

```

<cfcomponent extends="Controller">
    <cffunction name="secretStuff">
        <cfset restrictAccess(>
    </cffunction>

    <cffunction name="evenMoreSecretStuff">
        <cfset restrictAccess(>
    </cffunction>

    <cffunction name="restrictAccess">
        <cfif cgi.remote_addr Does Not Contain 12.55">
            <cfset flashInsert(msg="Sorry, we're not open in that area")>
            <cfset redirectTo(action="sorry")>
        <cfelseif NOT StructKeyExists(session,"userId")>
            <cfset flashInsert(msg="Please login first!")>
            <cfset redirectTo(action="login")>
        </cfif>
    </cffunction>
</cfcomponent>

```

Much better! But Wheels can take this process of avoiding repetition one step further. By placing a `filters()` call in the `init()` method of the controller, you can tell Wheels what function to run before any desired action(s).

```

<cfcomponent extends="Controller">
    <cffunction name="init">
        <cfset filters("restrictAccess")>
    </cffunction>

    <cffunction name="secretStuff">
    </cffunction>

    <cffunction name="evenMoreSecretStuff">
    </cffunction>

    <cffunction name="restrictAccess">
        <cfif cgi.remote_addr Does Not Contain 12.55">
            <cfset flashInsert(msg="Sorry, we're not open in that area")>

```

```
        <cfset redirectTo(action=sorry)>
    <cfelseif NOT StructKeyExists(session#userId)>
        <cfset flashInsert(msg=Please login first!)>
        <cfset redirectTo(action=login)>
    </cfif>
</cffunction>

</cfcomponent>
```

Besides the advantage of not having to call `restrictAccess()` twice, you have also gained two other things:

- The developer coding `secretStuff()` and `evenMoreSecretStuff()` can now focus on the main tasks of those two actions without having to worry about repetitive logic like authentication.
- The `init()` method is now starting to act like an overview for the entire controller.

All of these advantages will become much more obvious as your applications grow. This was just a simple example to put filters into context.

Sharing Filters Between Controllers

So far, we've only been dealing with one controller. Unless you're building a very simple website, you'll end up with a lot more.

The question then becomes, "Where do I place the `restrictAccess()` function so I can call it from any one of my controllers?" The answer is that because all controllers extend `Controller.cfc`, you should probably put it there. The `init()` method itself with the call to `filters()` should remain inside your individual controllers though.

If you actually want to set the same filters to be run for all controllers, you can go ahead and move it to the `Controller.cfc` file's `init()` method as well. Keep in mind that if you want to run the `init()` method in the individual controller and in `Controller.cfc`, you will need to call `super.init()` from the `init()` method of your individual controller.

2 Types of Filters

You specify if you want to run the filter function *before* or *after* the controller action with the `type` argument to the `filters()` function. It defaults to running it *before* the action.

The previous example with authentication showed a "before filter" in action. The other type of filter you can run is an "after filter." As you can tell from the name, an after filter executes code after the action has been completed.

This can be used to make some last minute modifications to the HTML before it is sent to the browser (think translation, compression, etc.), for example.

If you want to get a copy of the content that will be rendered to the browser from an after filter, you can use the `response()` function. To set your changes to the response afterward, use the `setResponse()` function.

As an example, let's say that you want to translate the content to Gibberish before sending it to your visitor. You can do something like this:

```
<cffunction name="init">
  <cfset filters(through="translate", type="after")>
</cffunction>

<cffunction name="translate">
  <cfset setResponse(gibberify(response()))>
</cffunction>
```

Including and Excluding Actions From Executing Filters

By default, filters apply to all actions in a controller. If that's not what you want, you can tell Wheels to only run the filter on the actions you specify with the `only` argument. Or you can tell it to run the filter on all actions except the ones you specify with the `except` argument.

Here are some examples showing how to setup filtering in your controllers. Remember, these calls go inside the `init()` function of your controller file.

```
<cfset filters(through="isLoggedIn,checkIPAddress"except="home,login")>
<cfset filters(through="translateText", only="termsOfUse", type="after")>
```

Passing Arguments to Filter Functions

Sometimes it's useful to be able to pass through arguments to the filters. For one, it can help you reduce the amount of functions you need to write. Here's the easy way to pass through an argument:

```
<cfset filters(through="authorize", byIP=true)>
```

Now the `byIP` argument will be available in the `authorize` function.

To help you avoid any clashing of argument names, Wheels also supports passing in the arguments in a struct as well:

```
<!--- The `through` argument would clash here if it weren't stored within a struct --->
<cfset args.byIP = true
<cfset args.through = true
<cfset filters(through="authorize", authorizeArguments=args)>
```

Evaluating Filter Arguments at Runtime

Because your controller's `init()` method only runs once per application start, the passing of arguments can also be written as expressions to be evaluated at runtime. This is helpful if you need for the value to be dynamic from request to request.

For example, this code would only evaluate the value for `request.region` on the very first request, and Wheels will store that particular value in memory for all subsequent requests:

```
<!--- This is probably not what you intended --->
<cfset filters(through="authorize", byIP=true, region=request.region)>
```

To avoid this "hard-coding" of values from request to request, you can instead pass an expression. (The double pound signs are necessary to escape dynamic values within the string. We only want to store a string representation of the expression to be evaluated.)

```
<!-- This is probably more along the lines of what you intended -->  
<cfset filters(through="authorize", byIP=true, region="##request.region##")>
```

Now instead of evaluating `request.region` inside the `init()` function, it will be done on each individual request.

Low Level Access

If you need to access your filters on a lower level, you can do so by using the `filterChain()` and `setFilterChain()` functions. Typically, you'll want to call `filterChain()` to return an array of all the filters set on the current controller, make your desired changes, and save it back using the `setFilterChain()` function.

Here is an example in our controller's `init()` method where we remove a certain restriction if the username is `perdjurner`:

```
<cfset var myFilterChain = filterChain()  
<cfif loggedInUser.username is perdjurner  
    <cfset StructDelete(myFilterChain,"restrictAccess")>  
</cfif>  
<cfset setFilterChain(myFilterChain)>
```

Verification

Verify the existence and type of variables before allowing for a given controller action to run.

Verification, through the `verifies()` function, is just a special type of filter that runs before actions. With verifications defined in your controller, you can eliminate the need for wrapping your entire actions in `<cfif>` blocks checking for the existence and types of variables. You also can limit your actions' scopes to specific request types like `post`, `get`, and AJAX requests.

Using `verifies()` to Enforce Request Types

Let's say that you want to make sure that all requests coming to a page that handles form submissions are `post` requests. While you can do this with a filter and the `isPost()` function, it is more convenient and DRY to do it with the `verifies()` function.

All that you need to do is add this line to your controller's `init()` function:

```
<!-- In the controller --->
<cffunction name="init">
    <cfset verifies(only="create,update" post=true)>
</cffunction>
```

The code above will ensure that all requests coming to the `create` and `update` actions are from form submissions. All other requests will be aborted.

There are also boolean arguments for `get` and `ajax` request types.

Defining a Handler for Failed Verifications

You can also specify different behavior for when the verification fails in a special handler function registered with the `handler` argument.

In this example, we register the `incorrectRequestType()` function as the handler:

```
<!-- In the controller --->
<cffunction name="init">
    <cfset verifies(
        only="create,update"
        post=true,
        handler="incorrectRequestType"
    )>
</cffunction>

<cffunction name="incorrectRequestType"
    <cfset redirectTo(action="accessDenied")>
</cffunction>
```

Note that you have to either do a `redirectTo()` call or abort the request completely after you've done what you wanted to do inside your handler function. If you don't do anything at all and just let the function exit on its own, Wheels will redirect the user back to the page they came from. In other words, you cannot render any content from inside this function but you can let another function handle that part by redirecting to it.

Enforcing the Existence and Type of Variables

A very convenient and common use of `verifies()` is when you want to make sure that a variables exists and is of a certain type; otherwise, you would like for your controller to redirect the user to a different page.

Step back in time for a moment and remember how you used to code websites before Wheels. (Yes I know those were dark days, but stay with me.)

On your `edit.cfm` page, what you probably did was write some code at the top of that looked like this:

```
<cfif !StructKeyExists(form,"userId") OR !IsValid("guid", form.userId)>
    <cflocation url="index.cfm" addToken=false>
</cfif>
```

With this snippet of code, you could ensure that any request to the `edit.cfm` had to have the `userId` in the form scope and that `userId` had to be of type `guid`. If these conditions weren't met, the request was redirected to the `index.cfm` page. This was a very tedious but necessary task.

Now let's see how using the `verifies()` function within Wheels improves this:

```
<!-- In the controller -->
<cffunction name="init">
    <cfset verifies(
        only="edit",
        post=true,
        params="userId",
        paramsTypes="guid",
        action="index",
        error="Invalid user ID."
    )>
</cffunction>
```

With that one line of code, Wheels will perform the following checks when a request is made to the controller's `edit` action:

- Make sure that the request is a `post` request.
- Make sure that the `userId` variables exists in the `params` struct.
- Make sure that the `params.userId` is of type `guid`.
- If any of those checks fail, redirect the request to the `index` action and place an error key in the Flash containing the message "Invalid user ID."

All that functionality in only one line of code!

What if you wanted to do this for two or more variables? The `params` and `paramsTypes` each accept a list so you can include as many variables to check against as you want.

The only thing you need to make sure of is that the number of variables in the `params` list matches the number types to check against in the `paramsTypes` list. This also goes for the `session/sessionTypes` and `cookie/cookieTypes` arguments, which check for existence and type in the `session` and `cookie` scopes, respectively.

Controller Verification vs. Model Object Validation

`verifies()` exists solely to validate *controller* and *environment level* variables and is not a substitute for Object Validation in your model.

A basic example of this is to validate params passed through to your controller from routes. Suppose we have the following route in our application:

```
<cfset addRoute(  
    name=#usersAddressesSave,  
    pattern#admin/users/[userid]/addresses/save"  
    controller#UserAddresses,"  
    action#save"  
)>
```

In this example, we will want to verify that the `userId` integer and `address` struct are both present in the `params` struct and also that `userId` is of a certain type:

```
<cfset verifies(  
    only=#save",  
    post=true,  
    params#userId,address,"  
    paramsTypes#integer,struct"  
)>
```

However, `verifies()` should not be used to make sure that values within the `address` struct themselves are valid (such as making sure that `address.zipCode` is correct). Because the `address` struct will be passed in to the model, the validation will be performed there.

Event Handlers

Use the standard CFML application events through the framework.

Because the `Application.cfc` file in the root of your Wheels site just includes the `wheels/functions.cfm` file, which in turn includes a lot of framework specific code, you may wonder what the best way is to use CFML's `onApplicationStart`, `onRequestStart`, etc. functions.

While it's perfectly possible to add your code directly to the `wheels/functions.cfm` file, we certainly don't recommend it. If you add code in there, you both increase the risk of accidentally modifying how the framework functions, and you also make it a lot harder to upgrade to future versions of Wheels.

Use the `events` Folder for Standard CFML Events

The general recommendation is to never touch any files in the `wheels` folder. OK, with that little warning out of the way, how does one go about using the CFML events?

The answer is to use the `events` folder. There is a file in there for every single event that CFML triggers. So if you want some code executed on application start for example, just place your code in `onapplicationstart.cfm`, and Wheels will run it when your application starts.

Wheels Includes Some Extra Bonus Events

If you look closely in the `events` folder, you will also notice that there are some custom files in there that do not match up with standard CFML events. The `onmaintenance.cfm` file is one example. Let's have a closer look at these.

On Maintenance

The `onmaintenance.cfm` file is included when Wheels is set to maintenance mode. After the file is included, `cfabort` is called by Wheels so no other code runs in this mode.

On Error

You can place a generic error message in the `onerror.cfm` file to be displayed to the users whenever your site throws an error.

If you need to access the error information here (for logging purposes, for example) it is available at `arguments.exception`.

On Missing Template

The `onmissingtemplate.cfm` file works in a similar way as the error file above, but it gets called when a controller or action in your application could not be found.

Note: If you want to make sure that all browsers show your custom 404 page you need to make it larger than 512 bytes in size. Google Chrome, for example, will display a friendly help page of its own when the 404 page is less than 512 bytes.

Adding Functions

Sometimes it's useful to add functions right in the `Application.cfc` file to make them available to all templates. To achieve the same thing in Wheels, you can place your functions in `events/functions.cfm`.

Application Settings

Again, because there is no `Application.cfc` file for you to work with in Wheels, you have to find a suitable place to set application settings such as `SessionManagement`, `SessionTimeout`, `ScriptProtect`, `SetClientCookies`, and so on. These are usually set in the constructor area of an `Application.cfc` file. We recommend that you set them in the `config/app.cfm` file instead.

URL Rewriting

Making URLs prettier using URL rewriting.

URL rewriting is a completely optional feature of Wheels, and all it does is get rid of the `index.cfm` part of the URL.

For example, with no URL rewriting, a URL in your application could look like this:

```
■ http://localhost/index.cfm/blog/new
```

After turning on URL rewriting, it would look like this:

```
■ http://localhost/blog/new
```

Combine this with the routing functionality of Wheels, and you get the capability of creating some really human-friendly (easier to remember, say over the phone, etc.) and search engine friendly (easier to crawl, higher PageRank, etc.) URLs.

Once you have uncommented the rewrite rules (found in either `.htaccess`, `web.config` or `IsapiRewrite4.ini`), Wheels will try and determine if your web server is capable of rewriting URLs and turn it on for you automatically. Depending on what web server you have and what folder you run Wheels from, you may need to tweak things a little though. Follow these instructions below for details on how to set up your web server and customize the rewrite rules when necessary.

Instructions for Apache

On most Apache setups, you don't have to do anything at all to get URL rewriting to work. Just uncomment the rewrite rules in the `.htaccess` file and Apache will pick up and use them automatically on server start-up.

There are some exceptions though...

If you have installed Apache yourself you may need to turn on the rewrite module and/or change the security settings before URL rewriting will work:

1. Check that the Apache `rewrite_module` has been loaded by ensuring there are no pound signs before the line that says `LoadModule rewrite_module modules/mod_rewrite.so` in the `httpd.conf` file.
2. Make sure that Apache has permission to load the rewrite rules from the `.htaccess` file. This is done by setting `AllowOverride` to `All` under the `Directory` section corresponding to the website you plan on using Wheels on (still inside the `httpd.conf` file).

If you have an older version of Apache and you're trying to run your Wheels site in a sub folder of an existing site you may need to hard code the name of this folder in your rewrite rules.

1. Change the last line of the `.htaccess` file to the following: `RewriteRule ^(.*)$ /sub_folder_name_goes_here/rewrite.cfm/$1 [L]`. Don't forget to change `sub_folder_name_goes_here` to the actual folder name first of course.

Instructions for IIS 7

Similar to Apache, IIS 7 will pick up the rewrite rules from a file located in the Wheels installation. In the case of IIS 7, the rules are picked up from the `web.config` file. (Don't forget to uncomment the XML block containing the rewrite rules in that file first.)

This requires that the URL Rewrite Module is installed. It's an IIS extension from Microsoft that you can download for free.

Instructions for IIS 6

Unfortunately, there is no built-in URL rewriting mechanism in IIS 6, so getting Wheels working with pretty URLs is a little more complicated than with Apache and IIS 7 (which often comes with the official "URL Rewrite Module" installed by default). Here's what you need to do:

1. Download Ionic's ISAPI Rewrite Filter. **NOTE:** the version must be v1.2.16 or later.
2. Unzip the file, get the `IsapiRewrite4.dll` file from the `lib` folder and put it in the root of your website. (It needs to be in the same folder as the `IsapiRewrite4.ini` file.)
3. To enable the rewrite filter in IIS 6, click on *Properties* for your website, then go to the *ISAPI Filters* tab and click the *Add...* button.
4. Type in anything you want as the *Filter Name* and point the *Executable* to the `IsapiRewrite4.dll` file.
5. Uncomment the rewrite rules in the `IsapiRewrite4.ini` file.

NOTE: Make sure you have "Verify that file exists" disabled for your site.

1. Right click your website and select *Properties*.
2. Click *Home Directory* tab.
3. Click the *Configuration* button.
4. Under the *Wildcard application mapping* section, double-click path for the `jrun_iis6_wildcard.dll`.
5. Uncheck *Verify that file exists*.
6. Click *OK* until all property screens are closed.

Deleting Unnecessary Files

The sole purpose of the `.htaccess` (for Apache), `web.config` (for IIS 7), and `IsapiRewrite4.ini` (for IIS 6) files is to make it possible to use URL rewriting.

Don't Forget to Restart

If you need to make changes to get URL rewriting to work, it's important to remember to always restart the web server and the ColdFusion server to make sure the changes are picked up by Wheels.

If you don't have access to restart services on your server, you can issue a `reload=true` request. It's often enough.

Using Routes

The convention for URLs in Wheels works for most situations and helps to promote an easy-to-maintain code base. With routes, you have the flexibility to break this convention when needed.

The Convention for URLs

To write clear MVC applications with ColdFusion on Wheels, we recommend sticking to conventions as much as possible. As you may already know, the convention for URLs is as follows:

```
■ http://www.domain.com/news/story/5
```

With this convention, the URL above tells Wheels to invoke the `story` action in the `news` controller. It also passes a parameter to the action called `key`, with a value of 5.

Creating Your Own Routes

But let's say that you wanted a simpler URL for your site's user profiles. What if you wanted to have a `profile` action in a controller called `user` with this URL?

```
■ http://www.domain.com/user/johndoe
```

Fear not, this is possible in Wheels.

Adding a New Route

Routes are configured in the `config/routes.cfm` file. This is where we'll add our new user profile route.

Routes are added to Wheels using the `addRoute()` function. Here is how we would set up our new route using `addRoute()`:

```
<cfset
    addRoute(
        name='userProfile', pattern='user/[username]',
        controller='user', action='profile'
    )
>
```

This call to `addRoute()` instructs Wheels to create a route named `userProfile` that passes a `username` parameter to the `profile` action in the `user` controller. This route will be invoked by any URL that starts with a top level folder of `user`. In most cases,

a pattern in a route should begin with a unique top level folder.

As you can see, any new parameters that you want to introduce to a new route should be surrounded by square brackets [].

With this, you can create URL patterns with any level of complexity that you wish.

Using controller and action Within Your Route

As you saw above, we specifically told Wheels which controller and action that the `userProfile` route should call. You also have the option of making the controller and action calls dynamic by including them in the pattern instead. (This is actually how Wheels sets up the default routes internally.)

You still have to pass in the `controller` and `action` arguments to the `addRoute()` function, but if `[controller]` and `[action]` also exist in the pattern, these arguments will only serve to tell Wheels what the default controller and action should be when they are not passed in through the URL.

Consider this line of code:

```
<cfset
  addRoute(
    name=adminUser, pattern=admin/user/[action],
    controller=adminUser, action=index"
  )
>
```

With this pattern, a URL that begins with `admin/user/` will always call the `adminUser` controller. But what action to call on that controller is determined dynamically by the URL.

But because we have `[action]` within the pattern, we don't always necessarily need for the route to point to `controllers/AdminUser.cfc's index()` method. The `action="index"` parameter instructs Wheels to call the `index` action when one is not otherwise specified in the URL.

So now this route pattern will, for example, match for these URLs:

URL	Controller	Action
<code>http://www.domain.com/admin/user/edit</code>	<code>adminUser</code>	<code>edit</code>
<code>http://www.domain.com/admin/user/add</code>	<code>adminUser</code>	<code>add</code>
<code>http://www.domain.com/admin/user/delete</code>	<code>adminUser</code>	<code>delete</code>

Although probably less useful, the same concept can be applied to the controller variable as well.

Linking to Your New Route

Now if you wanted to create a link to that user profile action we discussed earlier in the chapter, you would use Wheels's `linkTo()` function like so:

```
#linkTo(route="userProfile", username="johndoe")#
```

As you can see, `linkTo()` accepts a `route` argument, which changes the function's expectations on which other arguments are passed. Because our `userProfile` route expects a `username` parameter, we would need to pass that.

You can read more about creating links in the chapter called [Linking Pages](#).

Order of Precedence

With the potential of your application requiring many custom routes, you may wonder which order that Wheels considers these new routes. The answer is that Wheels gives precedence to the first listed custom route in your `config/routes.cfm` file.

Wheels will look through each custom route in order to see if there is a match. If not, it defaults to the default route mentioned at the beginning of this chapter under "The Convention for URLs".

Example of Precedence

Let's pretend that our `config/routes.cfm` file looks like this:

```
<cfset addRoute(name="newsAdmin", pattern="admin/news/[action]", controller="newsAdmin")>
<cfset addRoute(name="searchAdmin", pattern="admin/search/[action]", controller="searchAdmin")>
<cfset addRoute(name="adminRoot", pattern="admin/[action]", controller="admin")>
```

Wheels would make sure that the URL didn't begin with `admin/news` or `admin/search` before calling the third route listed, `adminRoot`.

If the URL didn't begin with `admin` at all, Wheels would use its internal default route, matching the usual pattern of `[controller]/[action]/[key]`.

Making a Catch-All Route

Sometimes you need a catch-all route in Wheels to support highly dynamic websites (like a content management system, for example), where all requests that are not matched by an existing route get passed to a controller/action that can deal with it.

Let's say you want to have both `http://localhost/welcome-to-the-site` and `http://localhost/terms-of-use` handled by the same controller and action. Here's what you can do to achieve this.

First, add a new route to `config/routes.cfm` that catches all pages like this:

```
<cfset addRoute(name="catchAll", pattern="[title]", controller="page", action="index")>
```

Now when you type in `http://localhost/welcome-to-the-site`, this route will be triggered and the `index` action will be called on the `page` controller with `params.title` set to `welcome-to-the-site`.

The problem with this is that this will break any of your normal controllers though, so you'll need to add them specifically before this route. (Remember the order of precedence explained above.)

You'll end up with a `config/routes.cfm` file looking something like this:

```
<cfset addRoute(name="main", pattern="main/[action]", controller="main")>  
<cfset addRoute(name="admin", pattern="admin/[action]", controller="admin")>  
<cfset addRoute(name="catchAll", pattern="[title]", controller="page", action="index")>  
<cfset addRoute(name="home", pattern="", controller="main", action="index")>
```

`main` and `admin` are your normal controllers. By adding them to the top of the routes file, Wheels looks for them first.

Obfuscating URLs

Hide your primary key values from nosy users.

The Wheels convention of using an auto-incrementing integer value as the primary key in your database tables will lead to a lot of URLs on your website exposing this value. Using the built-in URL obfuscation functionality in Wheels, you can hide this value from nosy users.

What URL Obfuscation Does

When URL obfuscation is turned off (which is the default setting in Wheels), this is how a lot of your URLs will end up looking:

```
■ http://localhost/user/profile/99
```

Here, 99 is the primary key value of a record in your users table.

After enabling URL obfuscation, this is how those URLs will look instead:

```
■ http://localhost/user/profile/b7ab9a50
```

The value 99 has now been obfuscated by Wheels to b7ab9a50. This makes it harder for nosy users to substitute the value to see how many records are in your users table, to name just one example.

How to Use It

To turn on URL obfuscation, all you have to do is call `set(obfuscateURLs=true)` in the `config/settings.cfm` file.

Once you do that, Wheels will handle everything else. Obviously, the main things Wheels does is obfuscate the primary key value when using the `linkTo()` function and deobfuscate it on the receiving end. Wheels will also obfuscate all other params sent in to `linkTo()` as well as any value in a form sent using a `get` request.

In some circumstances, you will need to obfuscate and deobfuscate values yourself if you link to pages without using the `linkTo()` function, for example. In these cases, you can use the `obfuscateParam()` and `deObfuscateParam()` functions to do the job for you.

Is This Really Secure?

No, this is not meant to add a high level of security to your application. It just obfuscates the values, making casual observation harder. It does not encrypt values, so keep that in mind when using this approach.

Caching

How to speed up your website by caching content.

If your website doesn't get a whole lot of traffic, then you can probably skip this chapter completely. Just remember that it's here, waiting for your triumphant return.

On the other hand, you're probably hoping for massive amounts of traffic to reach your website very soon, an imminent sell-out to Google, and a good life drinking Margaritas in the Caribbean. Knowing a little bit about the Wheels caching concepts will prepare you for this. (Except for the Margaritas... You're on your own with those ;).)

Consider a typical page on a Wheels website. It will likely call a controller action, get some data from your database using a finder method, and render the view for the user using some of the handy built-in functions.

All this takes time and resources on your web and database servers. Let's assume this page is accessed frequently, rarely changes, and looks the same for all users. Then you have a perfect candidate for caching.

Configuring the Cache

Wheels will configure all caching parameters behind the scenes for you based on what environment mode you are currently in. If you leave everything to Wheels, caching will be optimized for the different environment modes (and set to have reasonable settings for cache size and culling). This means that all caching is off when working in Design mode but on when in Production mode, for example.

Here are the global caching parameters you can set, their default values, and a description of what they do. Because these are not meant to be set differently based on the environment mode, you would usually set these in the `config/settings.cfm` file:

```
<cfset set(maximumItemsToCache=5000)
<cfset set(cacheCullPercentage=10)
<cfset set(cacheCullIntervalMinutes=5)
<cfset set(cacheDatePart="n")>
<cfset set(defaultCacheTime=60)
<cfset set(cacheQueriesDuringRequest=true)
<cfset set(clearQueryCacheOnReload=true)
<cfset set(cachePlugins=true)
```

maximumItemsToCache Setting

The total amount of items the cache can hold. When the cache is full, items will be deleted automatically at a regular interval based on the other settings below.

Note that the cache is stored in ColdFusion's `application` scope, so take this into consideration when deciding the number of items to store.

cacheCullPercentage Setting

This is the percentage of items that are culled from the cache when `maximumItemsToCache` has been reached.

For example, if you set this value to `10`, then at most, 10% of expired items will be deleted from the cache.

If you set this to `100`, then all expired items will be deleted. Setting it to `100` is perfectly fine for small caches but can be problematic if the cache is very large.

`cacheCullInterval` Setting

This is the number of minutes between each culling action. The reason the cache is not culled during each request is to keep performance as high as possible.

`cacheDatePart` Setting

The measurement unit to use during caching. The default is minutes ("`n`"), but you can set it to seconds ("`s`") for example if you want more precise control over when a cached item should expire. For the rest of this chapter, we'll assume you've left it at the default setting.

`defaultCacheTime` Setting

The number of minutes an item should be cached when it has not been specifically set through one of the functions that perform the caching in Wheels (i.e., `caches()`, `findAll()`, `includePartial()`, etc).

`cacheQueriesDuringRequest` Setting

When set to `true`, Wheels caches identical queries during a request. See the section titled *Automatic Caching of Identical Queries* below for more information about this setting.

`clearQueryCacheOnReload` Setting

When set to `true`, Wheels will clear out all cached queries when doing a `reload=true` request. This is usually a good idea, but if you want to avoid hitting the database too hard on application reloads, you can set this to `false` to keep queries cached and ease the load on the database.

`cachePlugins` Setting

If you set this to `false`, all plugins will reload on each request (as opposed to during `reload=true` requests only). Setting this is only recommended if you are developing a plugin yourself and need to see the impact of your changes as you develop it on a per-request basis.

Environment-Level Caching

The following cache variables are usually set per environment mode:

```
<cfset set(cacheDatabaseSchema=false)
<cfset set(cacheFileChecking=false)
<cfset set(cacheImages=false)
<cfset set(cacheModelInitialization=false)
<cfset set(cacheControllerInitialization=false)
<cfset set(cacheRoutes=false)
<cfset set(cacheActions=false)
<cfset set(cachePages=false)
<cfset set(cachePartials=false)
<cfset set(cacheQueries=false)
```

The settings shown above are what's in use in the Design mode. As you can see, when running in that mode, nothing is cached. This makes it possible to do database changes without having to issue a `reload=true` request or restart the ColdFusion server, for example. The downside is that it makes for slightly slower development because the pages will not load as fast in this environment.

The variables themselves are fairly self-explanatory. When `cacheDatabaseSchema` is set to `false`, you can add a field to the database, and Wheels will pick that up right away. When `cacheModelInitialization` is `false`, any changes you make to the `init()` function in the model file will be picked up. And so on.

Please refer to the Configuration and Defaults chapter for a complete listing of all the variables you can set and their default values.

4 Ways to Cache

In Wheels, you can cache data in 4 different ways:

- **Action caching** is the most effective of these methods because it caches the entire resulting HTML for the user.
- **Page caching** is similar to action caching, but it will only cache the view page itself and in reality, this caching method is rarely used.
- **Partial caching** is used when you only want to cache specific parts of pages. One reason for this could be that the page is personalized for a specific user, and you can only cache the sections that are not personalized.
- **Query caching** is the least effective of the 4 caching options because it only caches result sets that you get back from the database. But if you have a busy database and you're not too concerned with leaving pages/partials uncached, this could be a good option for you.

Action Caching

This code specifies that you want to cache the `browseByUser` and `browseByTitle` actions for 30 minutes:

```
<cfcomponent extends="Controller">
    <cffunction name="init">
```

```

        <cfset caches(actions=browseByUser,browseByTitle,time=30)>
    </cffunction>

    <cffunction name=browseByUser>
    </cffunction>

    <cffunction name=browseByTitle>
    </cffunction>

</cfcomponent>

```

As you can see, the `caches()` function call goes inside the `init()` function at the top of your controller file and accepts a list of action names and the number of minutes the actions should be cached for.

So what happens when users request the `browseByUser` page?

When Wheels receives the first request for this page, it will handle everything as it normally would, with the exception that it also adds a copy of the resulting HTML to the cache before ending the processing.

Wheels creates an internal key for use in the cache as well. This key is created from the `controller`, `action`, `key`, and `params` variables. This means, for example, that paginated pages are all stored individually in the cache (because the URL variable for the page to display would be different on each request).

When the second user requests the same page, Wheels will serve the HTML directly from the cache.

All subsequent requests now get the cached page until it expires.

But there are 2 exceptions to this (which you can make good use of in your code to have the cache re-created at the right times). If the request is a post request (normally coming from a form submission) or if the Flash (you can read everything about the Flash in the [Using the Flash](#) chapter) is not empty, then the cache won't be used. Instead, a new fresh page will be created.

One way to use this feature is to submit your forms to the same page to have it re-created or redirect to the cached page with a message in the Flash.

Here is some code that shows this technique with using the Flash to expire the cache. (Imagine that the `showArticle` page is cached and a user is adding a new comment to it.)

```

<cfset flashInsert(message="Your comment was added">
<cfset redirectTo(action="showArticle", key=params.key)>

```

Note that by default, any filters set for the action are being run as normal. This means that if you do authentication in the filter (which is a common technique for sites with content where you have to login first to see it), you can still cache those pages safely using the `caches()` function.

However, to achieve the fastest possible cache, you can override this default and tell Wheels to cache the HTML and serve that exactly as it is to all subsequent requests without running any filters or application events (`OnSessionStart`, `OnRequestStart`, etc.). To do this, set the `static` argument on `caches()` to `true`. This will cache your content using the `cfcache` tag behind the scenes. This means that the Wheels framework won't even get involved with the subsequent requests until they expire from the cache again.

Page Caching

This code specifies that you want to cache the view page for the `browseByUser` action for 1 hour:

```
<cfcomponent extends="Controller">
    <cffunction name="browseByUser">
        <cfset renderPage(cache=60)>
    </cffunction>

    <cffunction name="browseByTitle">
    </cffunction>
</cfcomponent>
```

The difference between action caching and page caching is that page caching will run the action and then only cache the view page itself. Action caching, as explained above, does not run the action code at all (but it does run filters and verifications).

Partial Caching

When your site contains personalized information (maybe some text specifying who you are logged in as, a list of items in your shopping cart, etc.), then action caching is not an option, and you need to cache at a lower level. This is where being able to cache only specific parts of pages comes in handy.

In Wheels, this is done by using the `cache` argument in a call to `includePartial()` or `renderPartial()`. You can pass in `cache=true` or `cache=x` where `x` is the number of minutes you want to cache the partial for.

If you just pass in `true`, the default cache expiration time will be used.

So, for example, if you have an e-commerce site that lists products with a shopping cart in the sidebar, then you'd create a partial for the list of products and cache only that.

Example code:

```
#includePartial(name="listing", cache=true)#
```

Query Caching

You can cache result sets returned by your queries too. As a ColdFusion developer, this probably won't be new to you because you've always been able to use the `cachedwithin` attribute in the `<cfquery>` tag. The query caching in Wheels is very similar to this.

You can use query caching on all finder methods, and it looks like this:

```
<cfset users = model('user').findAll(where=name LIKE 'a%', cache=10)>
```

So there you have it: 4 easy ways to speed up your website!

Automatic Caching of Identical Queries

When working with objects in Wheels, you'll likely find yourself using all of the convenient association functions Wheels makes available to you.

For example, if you have set up a `belongsTo` association from `article` to `author`, then you will likely write a lot of `article.author().name()` calls. In this case, Wheels is smart enough to only call the database once per request if the queries are identical. So don't worry about adding performance hits when making multiple calls like that in your code.

You can turn off this functionality either by using the `reload` argument to `findAll()` (or any of the dynamic methods that end up calling `findAll()` behind the scenes) or globally by adding `set(cacheQueriesDuringRequest=false)` to your configuration files.

Object Relational Mapping

An overview of Object Relational Mapping (ORM) and how it is used in Wheels. Learn how ORM simplifies your database interaction code.

Mapping objects in your application to records in your database tables is a key concept in Wheels. Let's take a look at exactly how this mapping is performed.

Class and Object Methods

Unlike most other languages, there is no notion of class level (a.k.a. "static") methods in ColdFusion. This means that even if you call a method that does not need to use any instance data, you still have to create an object first.

In Wheels, we create an object like this:

```
<cfset model("author")>
```

The built-in Wheels `model()` function will return a reference to an `author` object in the application scope (unless it's the first time you call this function, in which case it will also create and store it in the application scope).

Once you have the `author` object, you can start calling class methods on it, like `findByKey()`, for example. `findByKey()` returns an instance of the object with data from the database record defined by the key value that you pass.

Obviously, `author` is just an example here, and you'll use the names of the `.cfc` files you have created in the `models` folder.

```
<cfset authorClass = model("author")>  
<cfset authorObject = authorClass.findByKey(1)>
```

For readability, this is usually combined into the following:

```
<cfset authorObject = model("author").findByKey(1)>
```

Now `authorObject` is an instance of the `Author` class, and you can call object level methods on it, like `update()` and `save()`.

```
<cfset authorObject.update(firstName="Joe")>
```

In this case, the above code updates `firstName` field of the `author` record with a primary key value of 1 to Joe.

Primary Keys

Traditionally in Wheels, a primary key is usually named `id`, it increments automatically, and it's of the `integer` data type. However, Wheels is very flexible in this area. You can setup your primary keys in practically any way you want to. You can use *natural*

keys (varchar, for example), *composite keys* (having multiple columns as primary keys), and you can name the key(s) whatever you want.

You can also choose whether the database creates the key for you (using auto-incrementation, for example) or create them yourself directly in your code.

What's best, Wheels will introspect the database to see what choices you have made and act accordingly.

Tables and Classes

Wheels comes with a custom built ORM. ORM stands for "Object-Relational Mapping" and means that tables in your relational database map to classes in your application. The records in your tables map to objects of your classes, and the columns in these tables map to properties on the objects.

To create a class in your application that maps to a table in your database, all you need to do is create a new class file in your `models` folder and make it extend the `Model.cfc` file.

```
<cfcomponent extends="Model">
</cfcomponent>
```

If you don't intend to create any custom methods in your class, you can actually skip this step and just call methods without having a file created. It will work just as well. As your application grows, you'll probably want to have your own methods though, so remember the `models` folder. That's where they'll go.

Once you have created the file (or deliberately chosen not to for now), you will have a bunch of methods available handle reading and writing to the `authors` table. (For the purpose of showing some examples, we will assume that you have created a file named `Author.cfc`, which will then be mapped to the `authors` table in the database).

For example, you can write the following code to get the author with the primary key of 1, change his first name, and save the record back to the database.

```
<cfset auth = model{author}.findByKey(1)>
<cfset auth.firstName = "Joe">
<cfset auth.save(>
```

This code makes use of the class method `findByKey()`, updates the object property in memory, and then saves it back to the database using the object method `save()`. We'll get back to all these methods and more later.

Table and CFC Naming

By default, a table name should be the plural version of the class name. So if you have an `Author.cfc` class, the table name should be `authors`.

To change this behavior you can use the `table()` method. This method call should be placed in the `init()` method of your class file.

So, for example, if you wanted for your `author` model to map to a table in your database named `tbl_authors`, you would add the

following code to the `init()` method:

```
<cfcomponent extends="Model">
  <cffunction name="init">
    <cfset table("tbl_authors")>
  </cffunction>
</cfcomponent>
```

Columns and Properties

Objects in Wheels have properties that correspond to the columns in the table that it maps to. The first time you call a method on a model (or every time if you're in `design` mode), Wheels will reflect on the schema inside the database for the table the class maps to and extract all the column information.

To keep things as simple as possible, there are no getters or setters in Wheels. Instead, all the properties are made available in the `this` scope.

If you want to map a specific property to a column with a different name, you can override the Wheels mapping by using the `property()` method like this:

```
<cfcomponent extends="Model">
  <cffunction name="init">
    <cfset property(name="firstName", column="tbl_auth_f_name")>
  </cffunction>
</cfcomponent>
```

Creating Records

How to create new objects and save them to the database.

In Wheels, one way to create objects that represent records in our table is by calling the `new()` class-level method.

```
<cfset newAuthor = model("author").new(>
```

We now have an empty `Author` object that we can start filling in properties for. These properties correspond with the columns in the `authors` database table, unless you have mapped them specifically to columns with other names (or mapped to an entirely different table).

```
<cfset newAuthor.firstName = "John">  
<cfset newAuthor.lastName = "Doe">
```

At this point, the `newAuthor` object only exists in memory. We save it to the database by calling its `save()` method.

```
<cfset newAuthor.save(>
```

Creating Based on a Struct

If you want to create a new object based on parameters sent in from a form request, the `new()` method conveniently accepts a struct as well. As we'll see later, when you use the Wheels form helpers, they automatically turn your form variables into a struct that you can pass into `new()` and other methods.

Given that `params.newAuthor` is a struct containing the `firstName` and `lastName` variables, the code below does the same as the code above (without saving it though).

```
<cfset newAuthor = model("author").new(params.newAuthor)
```

Saving Straight to the Database

If you want to save a new author to the database right away, you can use the `create()` method instead.

```
<cfset model("author").create(params.newAuthor)
```

The Primary Key

Note that if we have opted to have the database create the primary key for us (which is usually done by auto-incrementing it), it will be available automatically after the object has been saved.

This means you can read the value by doing something like this. (This example assumes you have an auto-incrementing `integer` column named `id` as the primary key.)

```
<cfset newAuthor = model("author").new( )>
<cfset newAuthor.firstName = "Joe">
<cfset newAuthor.lastName = "Jones">
<cfset newAuthor.save( )>
<cfoutput#newAuthor.id#/cfoutput>
```

Don't forget that you can name your primary key whatever you want, and you can even use composite keys, natural keys, non auto-incrementing, and so on.

No matter which method you prefer, Wheels will use database introspection to see how your table is structured and act accordingly.

Using Database Defaults

The best way of handling model defaults is usually by setting a default constraint in your database. When Wheels saves the model to the database, it will automatically insert the default value if you haven't provided one within your model.

However, unlike the primary key, Wheels will not automatically load database defaults after saving as it requires an additional database call and in most cases is not required. (After saving, the most common action is to redirect, in which case you would reload the newly saved model in the next request anyway.)

Of course, if you do need to access the database default immediately after saving, Wheels allows this. Simply add `reload=true` to the `create()`, `update()`, or `save()` methods:

```
<cfset newAuthor = model("author").new( )>
<cfset newAuthor.firstName = "Joe">
<cfset newAuthor.lastName = "Jones">
<cfset newAuthor.save(reload=true)>
```

Using Model Defaults

Sometimes a database default isn't the most appropriate solution because the value is only set after the model has been inserted. If you want to set a default value when it is first created with `new()` or `create()`, then you can pass the `defaultValue` argument of the `property()` method used in your model's `init()` block.

```
<cfset property(name="welcomeText", defaultValue="Hello world!")>
```

This is effectively the same as doing this:

```
<cfset model("myModel").new(welcomeText="Hello world!")>
```

..except you only need to set it once per model.

Reading Records

Returning records from your database tables as objects or queries.

Reading records from your database typically involves using one of the 3 finder methods available in Wheels: `findByKey()`, `findOne()`, and `findAll()`.

The first 2 of these, `findByKey()` and `findOne()`, returns an object, while the last one, `findAll()`, returns the result from a `cfquery` tag.

Fetching a Row by Primary Key Value

Let's start by looking at the simplest of the finder methods, `findByKey()`. This method takes one argument: the primary key (or several keys if you're using composite keys) of the record you want to get.

If the record exists, it is returned to you as an object. If not, Wheels will return the boolean value `false`.

In the following example, we assume that the `params.key` variable has been created from the URL (for example a URL such as `http://localhost/blog/viewauthor/7.`)

In your controller:

```
<cfset author = model("author").findByKey(params.key)>
<cfif NOT IsObject(author)>
  <cfset flashInsert(message="Author #params.key# was not found")>
  <cfset redirectTo(back=true)>
</cfif>
```

In your view:

```
<cfoutput>Hello, #author.firstName# #author.lastName#!</cfoutput>
```

Fetching a Row by a Value Other Than the Primary Key

Often, you'll find yourself wanting to get a record (or many) based on a criteria other than just the primary key value.

As an example, let's say that you want to get the last order made by a customer. You can achieve this by using the `findOne()` method like so:

```
<cfset anOrder = model("order").findOne(order="datePurchased DESC")>
```

Fetching Multiple Rows

You can use `findAll()` when you are asking to get one or more records from the database. Wheels will return this as a `cfquery` result (which could be empty if nothing was found based on your criteria).

Arguments for `findOne()` and `findAll()`

Besides the difference in the default return type, `findOne()` and `findAll()` accept the same arguments. Let's have a closer look at these arguments.

select Argument

This maps to the `SELECT` clause of the SQL statement.

Wheels is pretty smart when it comes to figuring out what to select from the database table(s). For example, if nothing is passed in to the `select` argument, Wheels will assume that you want all columns returned and create a `SELECT` clause looking something like this:

```
■ artists.id,artists.name
```

As you can see, Wheels knows that the `artist` model is mapped to the `artists` table and will prepend the table name to the column names accordingly.

If you have mapped columns to a different property name in your application, Wheels will take this into account as well. The end result then could look like this:

```
■ artists.id,artists.fname AS firstName
```

If you select from more than one table (see the `include` argument below) and there are ambiguous column names, Wheels will sort this out for you by prepending the model name to the column name.

Let's say you have a column called `name` in both the `artists` and `albums` tables. The `SELECT` clause will be created like this:

```
■ artists.name,albums.name AS albumName
```

If you use the `include` argument a lot, you will love this feature as it saves a lot of typing.

If you don't want to return all properties, you can override this behavior by passing in a list of the properties you want returned.

If you want to take full control over the `SELECT` clause, you can do so by specifying the table names (i.e. `author.firstName`) in the `select` argument or by using alias names (i.e., `firstname AS firstName`). If Wheels comes across the use of any of these techniques, it will assume you know what you're doing and pass on the `select` argument straight to the `SELECT` clause with no changes.

A tip is to turn on debugging when you're learning Wheels so you can get a good understanding of how Wheels creates the SQL statements.

where Argument

This maps to the `WHERE` clause of the SQL statement. Wheels will also convert all your input to `cfqueryparam` tags for you automatically.

There are some limitations to what you can use in the `where` argument, but the following SQL will work: `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`, `LIKE`, `NOT LIKE`, `IN`, `NOT IN`, `IS NULL`, `IS NOT NULL`, `AND`, and `OR`. (Note that it's a requirement to write SQL keywords in upper case.) In addition to this, you can use parentheses to group conditional SQL statements together.

It's worth mentioning that although Wheels does not support the `BETWEEN` operator, you can get around this by using `>=` and `<=`.

Example:

```
<cfset items = model('item').findAll(where#price >= 100 AND price <= 500)>
```

The same goes for `NOT BETWEEN`:

```
<cfset items = model('item').findAll(where#price <= 100 OR price = 500)>
```

order Argument

This maps to the `ORDER` clause of the SQL statement. If you don't specify an order at all, none will be used. (Makes sense, eh?) So in those cases, the database engine will decide in what order to return the records. Note that it's a requirement to write the SQL keywords `ASC` and `DESC` in upper case.

There is one exception to this. If you paginate the records (by passing in the `page` argument) without specifying the order, Wheels will order the results by the primary key column. This is because pagination relies on having unique records to order by.

include Argument

This is a powerful feature that you can use if you have set up associations in your models.

If, for example, you have specified that one `Author` has many `Articles`, then you can return all authors and articles in the same call by doing this:

```
<cfset bobsArticles = model('author').findAll(where#firstName='Bob', include#Articles)>
```

maxRows Argument

This limits the number of records to return. Please note that if you call `findAll()` with `maxRows=1`, you will still get a `cfquery` result back and not an object. (We recommend using `findOne()` in this case if you want for an object to be returned.)

page and perPage Arguments

Set these if you want to get paginated data back.

So if you wanted records 11-20, for example, you write this code:

```
<cfset
  bobsArticles = model{author}.findAll(
    where=#firstName='Bob'," include=#Articles", page=2, perPage=10
  )
>
```

cache Argument

This is the number of minutes to cache the query for. This is eventually passed on to the `cachedwithin` attribute of the `cfquery` tag.

returnAs Argument

In the beginning of this chapter, we said that you either get a query or an object back depending on the method that you call. But you can actually specify the return type so that you get either an object, a query, or an array of objects back.

To do this, you use the `returnAs` argument. If you want an array of objects back from a `findAll()` call, for example, you can do this:

```
<cfset users = model{user}.findAll(returnAs=objects)>
```

On `findOne()` and `findByKey()`, you can set this argument to either `object` or `query`. On the `findAll()` method, you can set it to `objects` (note the plural) or `query`.

We recommend sticking to this convention as much as possible because of the CFML engines' slow `CreateObject()` function. Be careful when setting `returnAs` to `objects`. You won't want to create a lot of objects in your array and slow down your application unless you absolutely need to.

Updating Records

Updating records in your database tables.

When you have created or retrieved an object, you can save it to the database by calling its `save()` method. This method returns `true` if the object passes all validations and the object was saved to the database. Otherwise, it returns `false`.

This chapter will focus on how to update records. Read the Creating Records chapter for more information about how to create new records.

A Practical Example

Let's start with an example of getting a blog post from the database, updating its title, and saving it back:

```
<cfset post = model("post").findByKey(33)>
<cfset post.title = "New version of Wheels just released">
<cfset post.save()>
```

You can also change the values of one or more properties and save them to the database in one single call using the `update()` method, like this:

```
<cfset post = model("post").findByKey(33)>
<cfset post.update(title="New version of Wheels just released")>
```

Updating Via struct Values

You can also pass in name/value pairs to `update()` as a struct. The main reason this method accepts a struct is to allow you to easily use it with forms.

This is how it would look if you wanted to update the properties for a post based on a submitted form.

```
<cfset post = model("post").findByKey(params.key)>
<cfset post.update(params.post)>
```

It's also possible to combine named arguments with a struct, but then you need to name the struct argument as `properties`.

Example:

```
<cfset post = model("post").findByKey(params.key)>
<cfset
    post.update(
        title="New version of Wheels just released", properties=params.post
    )
>
```

Combine Reading and Updating into a Single Call

To cut down even more on lines of code, you can also combine the reading and saving of the objects by using the class-level methods `updateByKey()` and `updateAll()`.

The `updateByKey()` Method

Give the `updateByKey()` method a primary key value (or several if you use composite keys) in the `key` argument, and it will update the corresponding record in your table with the properties you give it. You can pass in the properties either as named arguments or as a struct to the `properties` argument.

This method returns the object with the primary key value you specified. If the object does not pass validation, it will be returned anyway, but nothing will be saved to the database.

By default, `updateByKey()` will fetch the object first and call the `update()` method on it, thus invoking any callbacks and validations you have specified for the model. You can change this behavior by passing in `instantiate=false`. Then it will just update the record from the table using a simple `UPDATE` query.

An example of using `updateByKey()` by passing a struct:

```
<cfset result = model("post").updateByKey(33, params.post)
```

And an example of using `updateByKey()` by passing named arguments:

```
<cfset
  result = model("post").updateByKey(
    id=33, title="New version of Wheels just released", published=1
  )
>
```

Updating Multiple Rows with `updateAll()`

The `updateAll()` method allows you to update more than one record in a single call. You specify what records to update with the `where` argument and tell Wheels what updates to make using named arguments for the properties.

The `where` argument is used exactly as you specify it in the `WHERE` clause of the query (with the exception that Wheels automatically wraps everything properly in `cfqueryparam` tags). So make sure that you place those commas and quotes correctly!

An example:

```
<cfset
  recordsReturned = model("post").updateAll(
    published=1, publishedAt=now(), where="published=0"
  )
>
```

Unlike `updateByKey()`, the `updateAll()` method will not instantiate the objects by default. That could be really slow if you wanted to update a lot of records at once.

Deleting Records

Deleting records from your database tables.

Deleting records in Wheels is simple. If you have fetched an object, you can just call its `delete()` method. If you don't have any callbacks specified for the class, all that will happen is that the record will be deleted from the table and `true` will be returned.

Delete Callbacks

If you have callbacks however, this is what happens:

First, all methods registered to be run before a delete happens (these are registered using a `beforeDelete()` call from the `init` function) will be executed, if any exist.

If these return `true`, Wheels will proceed and delete the record from the table. If `false` is returned from the callback code, processing will return to your code without the record being deleted. (`false` is returned to you in this case.)

If the record was deleted, the `afterDelete()` callback code is executed, and whatever that code returns will be returned to you. (You should make all your callbacks return `true` or `false`.)

If you're unfamiliar with the concept of callbacks, you can read about them in the Object Callbacks chapter.

Example of Deleting a Record

Here's a simple example of fetching a record from the database and then deleting it.

```
<cfset aPost = model("post").findByKey(33)>  
<cfset aPost.delete(>
```

There are also 3 class-level delete methods available: `deleteByKey()`, `deleteOne()`, and `deleteAll()`. These work similarly to the class level methods for updating, which you can read more about in [Updating Records](#).

Column Statistics

Use Wheels to get statistics on the values in a column, like row counts, averages, highest values, lowest values, and sums.

Since Wheels simplifies so much for you when you select, insert, update, and delete rows from the database, it would suck if you had to revert back to using `cfquery` and `COUNT(id) AS x` type queries when you wanted to get aggregate values, right?

Well, good news. Of course, you don't need to do this; just use the built-in functions `sum()`, `minimum()`, `maximum()`, `average()` and `count()`.

Let's start with the `count()` function, shall we?

Counting Rows

To count how many rows you have in your `authors` table, simply do this:

```
<cfset authorCount = model("author").count() >
```

What if you only want to count authors with a last name starting with "A"? Like the `findAll()` function, `count()` will accept a `where` argument, so you can do this:

```
<cfset authorCount = model("author").count(where="lastName LIKE 'A%'") >
```

Simple enough. But what if you wanted to count only authors in the USA, and that information is stored in a different table? Let's say you have stored country information in a table called `profiles` and also setup a `hasOne/belongsTo` association between the `author` and `profile` models.

Just like in the `findAll()` function, you can now use the `include` argument to reference other tables.

In our case, the code would end up looking something like this:

```
<cfset
  authorCount = model("author").count(
    include="profile", where="countryId=1 AND lastName LIKE 'A%'")
  >
```

Or, if you care more about readability than performance, why not just join in the `countries` table as well?

```
<cfset
  authorCount = model("author").count(
    include="profile(country), countries", where="name='USA' AND lastName LIKE 'A%'")
  >
```

In the background, these functions all perform SQL that looks like this:

```
SELECT COUNT(*)
FROM authors
WHERE ...
```

However, if you include a `hasMany` association, `Wheels` will be smart enough to add the `DISTINCT` keyword to the SQL. This makes sure that you're only counting unique rows.

For example, the following method call:

```
<cfset
  authorCount = model("author").count(
    include="books", where="title LIKE 'Wheels%'"
  )
>
```

Will execute this SQL (presuming `id` is the primary key of the `authors` table and the correct associations have been setup):

```
SELECT COUNT(DISTINCT authors.id)
FROM authors LEFT OUTER JOIN books ON authors.id = books.authorId
WHERE ...
```

OK, so now we've covered the `count()` function, but there are a few other functions you can use as well to get column statistics.

Getting an Average

You can use the `average()` function to get the average value on any given column. The difference between this function and the `count()` function is that this operates on a single column, while the `count()` function operates on complete records. Therefore, you need to pass in the name of the property you want to get an average for.

The same goes for the remaining column statistics functions as well; they all accept the `property` argument.

Here's an example of getting the average salary in a specific department:

```
<cfset
  avgSalary = model("employee").average(
    property="salary", where="departmentId=1"
  )
>
```

You can also pass in `distinct=true` to this function if you want to include only each unique instance of a value in the average calculation.

Getting the Highest and Lowest Values

To get the highest and lowest values for a property, you can use the `minimum()` and `maximum()` functions.

They are pretty self explanatory, as you can tell by the following examples:

```
<cfset highestSalary = model('employee').maximum('salary')>  
<cfset lowestSalary = model('employee').minimum('salary')>
```

Getting the Sum of All Values

The last of the column statistics functions is the `sum()` function.

As you have probably already figured out, `sum()` adds all values for a given property and returns the result. You can use the same arguments as with the other functions (`property`, `where`, `include`, and `distinct`).

Let's wrap up this chapter on a happy note by getting the total dollar amount you've made:

```
<cfset howRichAmI = model('invoice').sum('billedAmount')>
```

Dynamic Finders

Make your model calls more readable by using dynamic finders.

With the recent introduction of `onMissingMethod()` in ColdFusion 8 (Thanks, Adobe!), we have been able to port over the concept of *dynamic finders* from Rails to Wheels.

The concept is simple. Instead of using arguments to tell Wheels what you want to do, you can use a dynamically-named method.

For example, the following code:

```
<cfset me = model('user').findOne(where#email='me@myself.com')>
```

Can also be written as:

```
<cfset me = model('user').findOneByEmail("me@myself.com")>
```

Through the power of `onMissingMethod()`, Wheels will parse the method name and figure out that the value supplied is supposed to be matched against the `email` column.

Dynamic Finders Involving More than One Column

You can take this one step further by using code such as:

```
<cfset me = model('user').findOneByUsernameAndPassword("bob,pass")>
```

In this case, Wheels will split the function name on the `And` part and determine that you want to find the record where the `username` column is "bob" and the `password` column is "pass".

When you are passing in two values, make special note of the fact that they should be passed in as a list to one argument and not as two separate arguments.

Works with `findAll()` too

In the examples above, we've used the `findOne()` method, but you can use the same approach on a `findAll()` method as well.

Passing in Other Finder Parameters

In the background, these dynamically-named methods just pass along execution to `findOne()` or `findAll()`. This means that you can also pass in any arguments that are accepted by those two methods.

The below code, for example, is perfectly valid:

```
■ <cfset users = model("user").findAllByState(value#NY", order="name", page=3 >
```

When passing in multiple arguments like above, you have to start naming them instead of relying on the order of the arguments though. When doing so, you need to name the argument `value` if you're passing in just one value and `values` if you're passing in multiple values in a list. In other words, you need to name it `values` when calling an `And` type dynamic finder.

```
■ <cfset
  users = model("user").findAllByCityAndState(
    values#Buffalo,NY", order="name", page=3
  )
>
```

Avoid the Word "And" in Database Column Names

Keep in mind that this dynamic method calling will break down completely if you ever name a column `firstandlastname` or something similar because Wheels will then split the method name incorrectly. So avoid using "And" in the column name if you plan on taking advantage of dynamically-named finder methods.

Getting Paginated Data

Improve database performance and simplify your user interface by using pagination.

If you searched for "coldfusion" on Google, would you want all results to be returned on one page? Probably not because it would take a long time for Google to first get the records out its index and then prepare the page for you. Your browser would slow to a halt as it tried to render the page. When the page would finally show up, it would be a pain to scroll through all those results.

Rightly so, Google uses pagination to spread out the results on several pages.

And in Wheels, it's really simple to do this type of pagination. Here's how:

1. Get records from the database based on a page number. Going back to the Google example, this would mean getting records 11-20 when the user is viewing the second results page. This is (mostly) done using the `findAll()` function and the `page` and `perPage` arguments.
2. Display the links to all the other pages that the user should be able to go to. This is done using the `paginationLinks()` function or using a lower-level function such as `paginationHasNext()`. (See the "Related" section below for more info.)

This chapter will deal with the first part: getting the paginated data. Please proceed to the chapter called Displaying Links for Pagination if you wish to learn how to output the page links in your view.

Learning by Example

Let's jump straight to an example:

```
<cfset authors = model("Author").findAll(page=2, perPage=25, order=lastName)>
```

That simple code will return authors 26-50 from the database, ordered by their last name.

What SQL statements are actually being executed depends on which database engine you use. (The MySQL adapter will use `LIMIT` and `OFFSET`, and the Microsoft SQL Server adapter will use `TOP` and some tricky sub queries.) Turn on debugging in the ColdFusion Administrator if you want to see exactly what's going on under the hood.

One important thing that you should be aware of is that pagination is done based on objects and not records. To illustrate what that means, we can expand on the above example a little:

```
<cfset  
  authorsAndBooks = model("Author").findAll(  
    include="Books", page=2, perPage=25, order=lastName  
  )  
>
```

Here, we tell Wheels that we also want to include any books written by the authors in the result. Since it's possible that an author has written many books, we can't know in advance how many records we'll get back (as opposed to the first example, where we know we will get 25 records back). If each author has written 2 books, for example, we will get 50 records back.

If you do want to paginate based on the books instead, all that you need to do is flip the `findAll()` statement around a little:

```
<cfset
  booksAndAuthors = modelBook.findAll(
    include=Author, page=2, perPage=25, order=lastName"
  )
>
```

Here, we call the `findAll()` function on the `Book` class instead, and thereby we ensure that the pagination is based on the books and not the authors. In this case, we will always get 25 records back.

That's all there is to it, really. The best way to learn pagination is to play around with it with debugging turned on.

Related

- [Displaying Links for Pagination](#)

Associations

Through some simple configuration, Wheels allows you to unlock some powerful functionality to use your database tables' relationships in your code.

Associations in Wheels allow you to define the relationships between your database tables. After configuring these relationships, doing pesky table joins becomes a trivial task. And like all other ORM functions in Wheels, this is done without writing a single line of SQL.

3 Types of Associations

In order to set up associations, you only need to remember 3 simple methods. Considering that the human brain only reliably remembers up to 7 items, we've left you with a lot of extra space. You're welcome. :)

The association methods should always be called in the `init()` method of a model that relates to another model within your application.

The `belongsTo` Association

If your database table contains a field that is a foreign key to another table, then this is where to use the `belongsTo()` function.

If we had a `comments` table that contains a foreign key to the `posts` table called `postid`, then we would have this `init()` method within our comment model:

```
<cfcomponent extends="Model">
    <cffunction name="init">
        <cfset belongsTo("post")>
    </cffunction>
</cfcomponent>
```

The `hasOne` and `hasMany` Associations

On the other side of the relationship are the "has" functions. As you may have astutely guessed, these functions should be used according to the nature of the model relationship.

At this time, you need to be a little eccentric and talk to yourself. Your association should make sense in plain English language.

An example of `hasMany`

So let's consider the `post / comment` relationship mentioned above for `belongsTo()`. If we were to talk to ourselves, we would say, "A post has many comments." And that's how you should construct your `post` model:

```
<cfcomponent extends="Model">
    <cffunction name="init">
        <cfset hasMany("comments")>
    </cffunction>
</cfcomponent>
```

You may be a little concerned because our model is called `comment` and not `comments`. No need to worry: Wheels understands the need for the plural in conjunction with the `hasMany()` method.

And don't worry about those pesky words in the English language that aren't pluralized by just adding an "s" to the end. Wheels is smart enough to know that words like "deer" and "children" are the plurals of "deer" and "child," respectively.

An Example of `hasOne`

The `hasOne()` association is not used as often as the `hasMany()` association, but it has its use cases. The most common use case is when you have a large table that you have broken down into two or more smaller tables (a.k.a. denormalization) for performance reasons or otherwise.

Let's consider an association between `user` and `profile`. A lot of websites allow you to enter required info such as name and email but also allow you to add optional information such as age, salary, and so on. These can of course be stored in the same table. But given the fact that so much information is optional, it would make sense to have the required info in a `users` table and the optional info in a `profiles` table. This gives us a `hasOne()` relationship between these two models: "A user *has one* profile."

In this case, our `profile` model would look like this:

```
<cfcomponent extends="Model">
    <cffunction name="init">
        <cfset belongsTo("user")>
    </cffunction>
</cfcomponent>
```

And our `user` model would look like this:

```
<cfcomponent extends="Model">
    <cffunction name="init">
        <cfset hasOne("profile")>
    </cffunction>
</cfcomponent>
```

As you can see, you do not pluralize "profile" in this case because there is only one profile.

By the way, as you can see above, the association goes both ways, i.e. a `user hasOne()` `profile`, and a `profile belongsTo()` a `user`. Generally speaking, all associations should be set up this way. This will give you the fullest API to work with in terms of the methods and arguments that Wheels makes available for you.

However, this is not a definite requirement. Wheels associations are completely independent of one another, so it's perfectly OK to

setup a `hasMany()` association without specifying the related `belongsTo()` association.

Dependencies

A dependency is when an associated model relies on the existence of its parent. In example above, a `profile` is dependent on a `user`. When you delete the `user`, you would usually want to delete the `profile` as well.

Wheels makes this easy for you. When setting up your association, simply add the argument `dependent` with one of the following values, and Wheels will automatically deal with the dependency.

```
<!--- Instantiates the `profile` model and calls its `delete()` method --->
<cfset hasOne(name=#profile", dependent=#delete")>

<!--- Quickly deletes the `profile` without instantiating it --->
<cfset hasOne(name=#profile", dependent=#deleteAll')>

<!--- Sets the `userId` of the profile to `NULL` --->
<cfset hasOne(name=#profile", dependent=#nullify")>
```

You can create dependencies on `hasOne()` and `hasMany()` associations, but not `belongsTo()`.

Database Table Setup

Like everything else in Wheels, we strongly recommend a default naming convention for foreign key columns in your database tables.

In this case, the convention is to use the singular name of the related table with `id` appended to the end. So to link up our table to the `employees` table, the foreign key column should be named `employeeid`.

Breaking the Convention

Wheels offers a way to configure your models to break this naming convention, however. This is done by using the `foreignKey` argument in your models' `belongsTo()` calls.

Let's pretend that you have a relationship between `author` and `post`, but you didn't use the naming convention and instead called the column `post_id`. (You just can't seem to let go of the underscores, can you?)

Your `post`'s `init()` method would then need to look like this:

```
<cfcomponent extends="Model">
    <cffunction name="init">
        <cfset belongsTo(name="author", foreignKey="post_id")>
    </cffunction>
</cfcomponent>
```

You can keep your underscores if it's your preference or if it's required of your application.

Leveraging Model Associations in Your Application

Now that we have our associations set up, let's use them to get some data into our applications.

There are a couple ways to join data via associations, which we'll go over now.

Using the `include` Argument in `findAll()`

To join data from related tables in our `findAll()` calls, we simply need to use the `include` argument. Let's say that we wanted to include data about the `author` in our `findAll()` call for `posts`.

Here's what that call would look like:

```
■ <cfset posts = model('post').findAll(include='author')>
```

It's that simple. Wheels will then join the `authors` table automatically so that you can use that data along with the data from `posts`.

Note that if you switch the above statement around like this:

```
■ <cfset authors = model('author').findAll(include='posts')>
```

Then you would need to specify "post" in its plural form, "posts." If you're thinking about when to use the singular form and when to use the plural form, just use the one that seems most natural.

If you look at the two examples above, you'll see that in example #1, you're asking for all posts including each post's **author** (hence the singular "author"). In example #2, you're asking for all authors and all of the **posts** written by each author (hence the plural "posts").

You're not limited to specifying just one association in the `include` argument. You can for example return data for `authors`, `posts`, and `bios` in one call like this:

```
■ <cfset authorsPostsAndComments = model('author').findAll(include='posts,bio')>
```

To include several tables, simply delimit the names of the models with a comma. All models should contain related associations, or else you'll get a mountain of repeated data back.

Joining Tables Through a Chain of Associations

When you need to include tables more than one step away in a chain of joins, you will need to start using parenthesis. Look at the following example:

```
■ <cfset commentsPostsAndAuthors = model('comment').findAll(include='post(author')>
```

The use of parentheses above tells Wheels to look for an association named `author` on the `post` model instead of on the `comment` model. (Looking at the `comment` model is the default behavior when not using parenthesis.)

Handling Column Naming Collisions

There is a minor caveat to this approach. If you have a column in both associated tables with the same name, Wheels will pick just one to represent that column.

In order to include both columns, you can override this behavior with the `select` argument in the finder functions.

For example, if we had a column named `name` in both your `posts` and `authors` tables, then you could use the `select` argument like so:

```
<cfset
  post = model("post").findAll(
    select=#posts.name, authors.id, authors.post_id, authors.name AS authorname"
    include=#author"
  )
>
```

You would need to hard-code all column names that you need in that case, which does remove some of the simplicity. There are always trade-offs!

Using Dynamic Shortcut Methods

A cool feature of Wheels is the ability to use *dynamic shortcut* methods to work with the models you have set up associations for. By dynamic, we mean that the name of the method depends on what name you have given the association when you set it up. By shortcut, we mean that the method usually delegates the actual processing to another Wheels method but gives you, the developer, an easier way to achieve the task (and makes your code more readable in the process).

As usual, this will make more sense when put into the context of an example. So let's do that right now.

Example: Dynamic Shortcut Methods for Posts and Comments

Let's say that you tell Wheels through a `hasMany()` call that a `post` *has many* comments. What happens then is that Wheels will enrich the `post` model by adding a bunch of useful methods related to this association.

If you wanted to get all `comments` that have been submitted for a `post`, you can now call `post.comments()`. In the background, Wheels will delegate this to a `findAll()` call with the `where` argument set to `postid=#post.id#`.

Listing of Dynamic Shortcut Methods

Here are all the methods that are added for the three possible association types.

Methods Added by `hasMany`

Given that you have told Wheels that a `post` *has many* comments through a `hasMany()` call, here are the methods that will be made available to you on the `post` model.

Replace `xxx` below with the name of the associated model (i.e. `comments` in the case of the example that we're using here).

Method	Example	Description
<code>xxx()</code>	<code>post.comments()</code>	Returns all <code>comments</code> where the foreign key matches the <code>post</code> 's primary key value. Similar to calling <code>model("comment").findAll(where="postid=#post.id#")</code> .
<code>addxxx()</code>	<code>post.addComment(comment)</code>	Adds a <code>comment</code> to the <code>post</code> association by setting its foreign key to the <code>post</code> 's primary key value. Similar to calling <code>model("comment").updateByKey(key=comment.id, postid=post.id)</code> .
<code>removexxx()</code>	<code>post.removeComment(comment)</code>	Removes a <code>comment</code> from the <code>post</code> association by setting its foreign key value to <code>NULL</code> . Similar to calling <code>model("comment").updateByKey(key=comment.id, postid="")</code> .
<code>deletexxx()</code>	<code>post.deleteComment(comment)</code>	Deletes the associated <code>comment</code> from the database table. Similar to calling <code>model("comment").deleteByKey(key=comment.id)</code> .
<code>removeAllxxx()</code>	<code>post.removeAllComments()</code>	Removes all <code>comments</code> from the <code>post</code> association by setting their foreign key values to <code>NULL</code> . Similar to calling <code>model("comment").updateAll(postid="", where="postid=#post.id#")</code> .
<code>deleteAllxxx()</code>	<code>post.deleteAllComments()</code>	Deletes the associated <code>comments</code> from the database table. Similar to calling <code>model("comment").deleteAll(where="postid=#post.id#")</code> .
<code>xxxCount()</code>	<code>post.commentCount()</code>	Returns the number of associated <code>comments</code> . Similar to calling <code>model("comment").count(where="postid=#post.id#")</code> .
<code>newxxx()</code>	<code>post.newComment()</code>	Creates a new <code>comment</code> object. Similar to calling <code>model("comment").new(postid=post.id)</code> .
<code>createxxx()</code>	<code>post.createComment()</code>	Creates a new <code>comment</code> object and saves it to the database. Similar to calling <code>model("comment").create(postid=post.id)</code> .
<code>hasxxx()</code>	<code>post.hasComments()</code>	Returns <code>true</code> if the <code>post</code> has any <code>comments</code> associated with it. Similar to calling <code>model("comment").exists(where="postid=#post.id#")</code> .
<code>findOnexxx()</code>	<code>post.findOneComment()</code>	Returns one of the associated <code>comments</code> . Similar to calling <code>model("comment").findOne(where="postid=#post.id#")</code> .

Methods Added by `hasOne`

The `hasOne()` association adds a few methods as well. Most of them are very similar to the ones added by `hasMany()`.

Given that you have told `Wheels` that an `author` has one `profile` through a `hasOne()` call, here are the methods that will be made available to you on the `author` model.

Method	Example	Description
<code>xxx()</code>	<code>author.profile()</code>	Returns the <code>profile</code> where the foreign key matches the <code>author</code> 's primary key value. Similar to calling <code>model("profile").findOne(where="authorid=#author.id#")</code> .
<code>setxxx()</code>	<code>author.setProfile(profile)</code>	Sets the <code>profile</code> to be associated with the <code>author</code> by setting its foreign key to the <code>author</code> 's primary key value. You can pass in either a <code>profile</code> object or the primary key value of a <code>profile</code> object to this method. Similar to calling <code>model("profile").updateByKey(key=profile.id, authorid=author.id)</code> .

removeXXX()	<code>author.removeProfile()</code>	Removes the <code>profile</code> from the <code>author</code> association by setting its foreign key to <code>NULL</code> . Similar to calling <code>model("profile").updateOne(where="authorid=#author.id#", authorid="")</code> .
deleteXXX()	<code>author.deleteProfile()</code>	Deletes the associated <code>profile</code> from the database table. Similar to calling <code>model("profile").deleteOne(where="authorid=#author.id#")</code> .
newXXX()	<code>author.newProfile()</code>	Creates a new <code>profile</code> object. Similar to calling <code>model("profile").new(authorid=author.id)</code> .
createXXX()	<code>author.createProfile()</code>	Creates a new <code>profile</code> object and saves it to the database. Similar to calling <code>model("profile").create(authorid=author.id)</code> .
hasXXX()	<code>author.hasProfile()</code>	Returns <code>true</code> if the <code>author</code> has an associated <code>profile</code> . Similar to calling <code>model("profile").exists(where="authorid=#author.id#")</code> .

Methods Added by `belongsTo`

The `belongsTo()` association adds a couple of methods to your model as well.

Given that you have told `Wheels` that a `comment` belongs to a `post` through a `belongsTo()` call, here are the methods that will be made available to you on the `comment` model.

Method	Example	Description
xxx()	<code>comment.post()</code>	Returns the <code>post</code> where the primary key matches the <code>comment</code> 's foreign key value. Similar to calling <code>model("post").findByKey(comment.postid)</code> .
hasXXX()	<code>comment.hasPost()</code>	Returns <code>true</code> if the <code>comment</code> has a <code>post</code> associated with it. Similar to calling <code>model("post").exists(comment.postid)</code> .

One general rule for all of the methods above is that you can always supply any argument that is accepted by the method that the processing is delegated to. This means that you can, for example, call `post.comments(order="createdAt DESC")`, and the `order` argument will be passed along to `findAll()`.

Another rule is that whenever a method accepts an object as its first argument, you also have the option of supplying the primary key value instead. This means that `author.setProfile(profile)` will perform the same task as `author.setProfile(1)`. (Of course, we're assuming that the `profile` object in this example has a primary key value of 1.)

Performance of Dynamic Association Finders

You may be concerned that using a dynamic finder adds yet another database call to your application.

If it makes you feel any better, all calls in your `Wheels` request that generate the same SQL queries will be cached for that request. No need to worry about the performance implications of making multiple calls to the same `author.posts()` call in the scenario above, for example.

Passing Arguments to Dynamic Shortcut Methods

You can also pass arguments to dynamic shortcut methods where applicable. For example, with the `xxx()` method, perhaps we'd want to limit a `post`'s comment listing to just ones created today. We can pass a `where` argument similar to what is passed to the `findAll()` function that powers `xxx()` behind the scenes.

```
<cfset today = DateFormatNow(), "yyyy-mm-dd">
<cfset comments = post.comments(where createdAt >= '#today# 00:00:00')>
```

Many-to-Many Relationships

We can use the same 3 association functions to set up many-to-many table relationships in our models. It follows the same logic as the descriptions mentioned earlier in this chapter, so let's jump right into an example.

Let's say that we wanted to set up a relationship between `customers` and `publications`. A customer can be subscribed to many publications, and publications can be subscribed to by many customers. In our database, this relationship is linked together by a third table called `subscriptions` (sometimes called a bridge entity by ERD snobs).

Setting up the Models

Here are the representative models:

```
<!--- Customer.cfc --->
<cfcomponent extends="Model">

    <cffunction name="init">
        <cfset hasMany("subscriptions")>
    </cffunction>

</cfcomponent>
```

```
<!--- Publication.cfc --->
<cfcomponent extends="Model">

    <cffunction name="init">
        <cfset hasMany("subscriptions")>
    </cffunction>

</cfcomponent>
```

```
<!--- Subscription.cfc --->
<cfcomponent extends="Model">

    <cffunction name="init">
        <cfset belongsTo("customer")>
        <cfset belongsTo("publication")>
    </cffunction>

</cfcomponent>
```

This assumes that there are foreign key columns in `subscriptions` called `customerid` and `publicationid`.

Using Finders with a Many-to-Many Relationship

At this point, it's still fairly easy to get data from the many-to-many association that we have set up above.

We can include the related tables from the `subscription` bridge entity to get the same effect:

```
■ <cfset data = model{subscription}.findAll(include="customer,publication")>
```

Creating a Shortcut for a Many-to-Many Relationship

With the `shortcut` argument to `hasMany()`, you can have Wheels create a dynamic method that lets you bypass the join model and instead reference the model on the other end of the many-to-many relationship directly.

For our example above, you can alter the `hasMany()` call on the `customer` model to look like this instead:

```
■ <cfset hasMany(name="subscriptions", shortcut="publications")>
```

Now you can get a customer's publications directly by using code like this:

```
■ <cfset cust = model{customer}.findByKey(params.key)
  <cfset pubs = cust.publications*>
```

This functionality relies on having set up all the appropriate `hasMany()` and `belongsTo()` associations in all 3 models (like we have in our example in this chapter).

It also relies on the association names being consistent, but if you have customized your association names, you can specify exactly which associations the shortcut method should use with the `through` argument.

The `through` argument accepts a list of 2 association names. The first argument is the name of the `belongsTo()` association (set in the `subscription` model in this case), and the second argument is the `hasMany()` association going back the other way (set in the `publication` model).

Sound complicated? That's another reason to stick to the conventions whenever possible: it keeps things simple.

Are You Still with Us?

As you just read, Wheels offers a ton of functionality to make your life easier in working with relational databases. Be sure to give some of these techniques a try in your next Wheels application, and you'll be amazed at how little code that you'll need to write to interact with your database.

Nested Properties

Save data in associated model objects through the parent.

When you're starting out as a Wheels developer, you are probably amazed at the simplicity of a model's CRUD methods. But then it all gets quite a bit more complex when you need to update records in multiple database tables in a single transaction.

Nested properties in Wheels makes this scenario dead simple. With a configuration using the `nestedProperties()` function in your model's `init()` method, you can save changes to that model and its associated models in a single call with `save()`, `create()`, or `update()`.

One-to-One Relationships with Nested Properties

Consider a user model that has one profile:

```
<!-- models/User.cfc -->
<cfcomponent extends="Model">

    <cffunction name="init">
        <cfset hasOne("profile")>
        <cfset nestedProperties(association="profile")>
    </cffunction>

</cfcomponent>
```

By adding the call to `nestedProperties()` in the model, you can create both the `user` object and the `profile` object in a single call to `create()`.

Setting up Data for the user Form in the Controller

First, in our controller, let's set the data needed for our form:

```
<!-- In controllers/User.cfc -->
<cffunction name="new">
    <cfset var newProfile = model("profile").new(>
        <cfset user = model("user").new(profile=newProfile)
    </cffunction>
```

Because our form will also expect an object called `profile` nested within the `user` object, we must create a new instance of it and set it as a property in the call to `user.new()`.

Also, because we don't intend on using the particular `newProfile` object set in the first line of the action, we can `var` scope it to clearly mark our intentions for its use.

If this were an `edit` action calling an existing object, our call would need to look similar to this:

```
<!-- In controllers/User.cfc --->
<cffunction name="edit">
    <cfset user = model("user").findByKey(key=params.key, include="profile")>
</cffunction>
```

Because the form will also expect data set in the `profile` property, you must include that association in the finder call with the `include` argument.

Building a Form for Posting Nested Properties

For this example, our form at `views/users/new.cfm` will end up looking like this:

```
#startFormTag(action="create")#

<!-- Data for user model --->
#textField(label="First Name", objectName="user", property="firstName")#
#textField(label="Last Name", objectName="user", property="lastName")#

<!-- Data for associated profile model --->
#textField(
    label="Twitter Handle",
    objectName="user",
    association="profile",
    property="twitterHandle"
)#
#textArea(
    label="Biography",
    objectName="user",
    association="profile",
    property="bio"
)#

<div>#submitTag(value="Create")#</div>

#endFormTag()#
```

Of note are the calls to form helpers for the `profile` model, which contain an extra argument for `association`. This argument is available for all object-based form helpers. By using the `association` argument, Wheels will name the form field in such a way that the properties for the profile will be nested within an object in the `user` model.

Take a minute to read that last statement again. OK, let's move on to the action that handles the form submission.

Saving the Object and Its Nested Properties

You may be surprised to find out that our standard `create` action does not change at all from what you're used to.

```
<!-- In controllers/Users.cfc --->
<cffunction name="create">
    <cfset user = model("user").new(params.user)>
    <cfif user.save()>
        <cfset flashInsert("success"The user was created successfully)">
```

```

        <cfset redirectTo(controller=params.controller)
    <cfelse>
        <cfset renderPage(action="new")>
    </cfif>
</cfunction>

```

When calling `user.save()` in the example above, `Wheels` takes care of the following:

- Saves the data passed into the `user` model.
- Sets a property on `user` called `profile` with the `profile` data stored in an object.
- Saves the data passed into that `profile` model.
- Wraps all calls in a transaction in case validations on any of the objects fail or something wrong happens with the database.

For the `edit` scenario, this is what our `update` action would look like (which is very similar to `create`):

```

<!-- In controllers/Users.cfc -->
<cfunction name="update">
    <cfset user = model("user").findByKey(params.user.id)
    <cfif user.update(params.user)>
        <cfset flashInsert(success="The user was updated successfully")>
        <cfset redirectTo(action="edit")>
    <cfelse>
        <cfset renderPage(action="edit")>
    </cfif>
</cfunction>

```

One-to-Many Relationships with Nested Properties

Nested properties work with one-to-many associations as well, except now the nested properties will contain an array of objects instead of a single one.

In the `user` model, let's add an association called `addresses` and also enable it as nested properties.

```

<!-- models/User.cfc -->
<cfcomponent extends="Model">

    <cfunction name="init">
        <cfsethasOne("profile")>
        <cfsethasmany("addresses")>
        <cfsetnestedproperties(
            association$profile,addresses,"
            allowDelete=true
        )>
    </cfunction>

</cfcomponent>

```

In this example, we have added the `addresses` association to the call to `nestedProperties()`.

Setting up Data for the user Form in the Controller

Setting up data for the form is similar to the one-to-one scenario, but this time the form will expect an array of objects for the nested properties instead of a single object.

In this example, we'll just put one new address in the array.

```
<!-- In controllers/Users.cfc -->
<cffunction name="new">
    <cfset var newAddresses = [ model(address).new() ] >
    <cfset user = model("user").new(addresses=newAddresses) >
</cffunction>
```

In the edit scenario, we just need to remember to call the `include` argument to include the array of addresses saved for the particular user:

```
<!-- In controllers/Users.cfc -->
<cffunction name="edit">
    <cfset user = model("user").findByKey(key=params.key, include=addresses) >
</cffunction>
```

Building the Form for the One-to-Many Association

This time, we'll add a section for addresses on our form:

```
#startFormTag(action="create")#

<!-- Data for `user` model -->
<fieldset>
    <legend>User</legend>

    #textField(label="First Name", objectName="user", property="firstName")#
    #textField(label="Last Name", objectName="user", property="lastName")#
</fieldset>

<!-- Data for `address` models -->
<fieldset>
    <legend>Addresses</legend>

    <div id="addresses">
        #includePartial(user.addresses)#
    </div>
</fieldset>

<div>#submitTag(value="Create")#</div>

#endFormTag()#
```

In this case, you'll see that the form for addresses is broken into a partial. (See the chapter on `Partials` for more details.) Let's take a look at that partial.

The `_address` Partial

Here is the code for the partial at `views/users/_address.cfm`. Wheels will loop through each address in your nested properties and display this piece of code for each one.

```
<div class=#address>
  #textField(
    label="Street",
    objectName="user",
    association="addresses",
    position=arguments.current,
    property="address1"
  )#
  #textField(
    label="City",
    objectName="user",
    association="addresses",
    position=arguments.current,
    property="city"
  )#
  #textField(
    label="State",
    objectName="user",
    association="addresses",
    position=arguments.current,
    property="state"
  )#
  #textField(
    label="Zip",
    objectName="user",
    association="addresses",
    position=arguments.current,
    property="zip"
  )#
</div>
```

Because there can be multiple addresses on the form, the form helpers require an additional argument for `position`. Without having a unique position identifier for each address, Wheels would have no way of understanding which `state` field matches with which particular address, for example.

Here, we're passing a value of `arguments.current` for `position`. This value is set automatically by Wheels for each iteration through the loop of addresses.

Auto-saving a Collection of Child Objects

Even with a complex form with a number of child objects, Wheels will save all of the data through its parent's `save()`, `update()`, or `create()` methods.

Basically, your typical code to save the `user` and its `addresses` is exactly the same as the code demonstrated in the *Saving the Object and Its Nested Properties* section earlier in this chapter.

Writing the action to save the data is clearly the easiest part of this process!

Transactions are Included by Default

As mentioned earlier, Wheels will automatically wrap your database operations for nested properties in a transaction. That way if something goes wrong with any of the operations, the transaction will rollback, and you won't end up with incomplete saves.

See the chapter on Transactions for more details.

Many-to-Many Relationships with Nested Properties

We all enter the scenario where we have "join tables" where we need to associate models in a many-to-many fashion. Wheels makes this pairing of entities simple with some form helpers.

Consider the many-to-many associations related to customers, publications, and subscriptions, straight from the Associations chapter.

```
<!-- models/Customer.cfc -->
<cfcomponent extends="Model">

    <cffunction name="init">
        <cfset hasMany(name="subscriptions", shortcut="publications")>
    </cffunction>

</cfcomponent>
```

```
<!-- models/Publication.cfc -->
<cfcomponent extends="Model">

    <cffunction name="init">
        <cfset hasMany("subscriptions")>
    </cffunction>

</cfcomponent>
```

```
<!-- models/Subscription.cfc -->
<cfcomponent extends="Model">

    <cffunction name="init">
        <cfset belongsTo("customer")>
        <cfset belongsTo("publication")>
    </cffunction>

</cfcomponent>
```

When it's time to save customers' subscriptions in the subscriptions join table, one approach is to loop through data submitted by `checkboxTag()`s from your form, populate subscription model objects with the data, and call `save()`. This approach is valid, but it is quite cumbersome. Fortunately, there is a simpler way.

Setting up the Nested Properties in the Model

Here is how we would set up the nested properties in the `customer` model for this example:

```
<!-- models/Customer.cfc -->
<cfcomponent extends="Model">

    <cffunction name="init">
        <!-- Associations -->
        <cfset hasMany(name="subscriptions" shortcut="publications")>
        <!-- Nested properties -->
        <cfset nestedProperties(
            association="subscriptions",
            allowDelete=true
        )
    </cffunction>
</cfcomponent>
```

Setting up Data for the `customer` Form in the Controller

Let's define the data needed in an edit action in the controller at `controllers/Customers.cfc`.

```
<!-- In `controllers/Customers.cfc` -->
<cffunction name="edit">

    <cfset customer = model("customer").findByKey(
        key=params.key,
        include="subscriptions"
    )
    <cfset publications = model("publication").findAll(order="title")>
</cffunction>
```

For the view, we need to pull the `customer` with its associated `subscriptions` included with the `include` argument. We also need all of the `publications` in the system for the user to choose from.

Building the Many-to-Many Form

We can now build a series of check boxes that will allow the end user choose which `publications` to assign to the `customer`.

The view template at `views/customers/edit.cfm` is where the magic happens. In this view, we will have a form for editing the `customer` and check boxes for selecting the `customer's` `subscriptions`.

```
<cfparam name="customer">
<cfparam name="publications" type="query">

<cfoutput>

#startFormTag(action="update")#

<fieldset>
    <legend>Customer</legend>
```

```

        #textField(
            label="First Name",
            objectName="customer",
            property="firstName"
        )#
        #textField(
            label="Last Name",
            objectName="customer",
            property="lastName"
        )#
    </fieldset>

    <fieldset>
        <legend>Subscriptions</legend>

        <cfloop query=#publications#>
            #hasManyCheckBox(
                label=publications.title,
                objectName="customer",
                association="subscriptions",
                keys="#customer.key()#,#publications.id#"
            )#
        </cfloop>
    </fieldset>

    <div>
        #hiddenField(objectName="customer", value="id")#
        #submitTag()#
    </div>

    #endFormTag()#

</cfoutput>

```

The main point of interest in this example is the `<fieldset>` for Subscriptions, which loops through the query of publications and uses the `hasManyCheckBox()` form helper. This helper is similar to `checkBox()` and `checkBoxTag()`, but it is specifically designed for building form data related by associations. (Note that `checkBox()` is primarily designed for columns in your table that store a single `true/ false` value, so that is the big difference.)

Notice that the `objectName` argument passed to `hasManyCheckBox()` is the parent `customer` object and the `associations` argument contains the name of the related association. Wheels will build a form variable named in a way that the `customer` object is automatically bound to the `subscriptions` association.

The `keys` argument accepts the foreign keys that should be associated together in the `subscriptions` join table. Note that these keys should be listed in the order that they appear in the database table. In this example, the `subscriptions` table in the database contains a composite primary key with columns called `customerid` and `publicationid`, in that order.

How the Form Submission Works

Handling the form submission is the most powerful part of the process, but like all other nested properties scenarios, it involves no extra effort on your part.

You'll notice that this example `update` action is fairly standard for a Wheels application:

```

<cffunction name="update">
    <!-- Load customer object -->
    <cfset customer = model("customer").findByKey(params.customer.id)
    <!-- If update is successful, generate success message
         and redirect back to edit screen -->
    <cfif customer.update(params.customer)>
        <cfset redirectTo(
            action="edit",
            key=customer.id,
            success="#customer.firstName# #customer.lastName# record updated successfully."
        )>
    <!-- If update fails, show form with errors -->
    <cfelse>
        <cfset renderPage(action="edit")>
    </cfif>
</cffunction>

```

In fact, there is nothing special about this. But with the nested properties defined in the model, Wheels handles quite a bit when you save the parent customer object:

- Wheels will update the `customers` table with any changes submitted in the Customers `<fieldset>`.
- Wheels will add and remove records in the `subscriptions` table depending on which check boxes are selected by the user in the Subscriptions `<fieldset>`.
- All of these database queries will be wrapped in a transaction. If any of the above updates don't pass validation or if the database queries fail, the transaction will roll back.

Object Validation

Wheels utilizes validation setup within the model to enforce appropriate data constraints and persistence. Validation may be performed for saves, creates, and updates.

Basic Setup

In order to establish the full cycle of validation, 3 elements need to be in place:

1. **Model** file containing business logic for the database table. Example: `models/User.cfc`
2. **Controller** file for creating, saving or updating a model instance. Example: `controllers/Users.cfc`
3. **View** file for displaying the original data inputs and an error list. Example: `views/users/index.cfm`

Note: Saving, creating, and updating model objects can also be done from the model file itself (or even in the view file if you want to veer completely off into the wild). But to keep things simple, all examples in this chapter will revolve around code in the controller files.

The Model

Validations are always defined in the `init()` method of your model. This keeps everything nice and tidy because another developer can check `init()` to get a quick idea on how your model behaves.

Let's dive right into a somewhat comprehensive example:

```
<cfcomponent extends="Model" output="false">
    <cffunction name="init">
        <cfset validatesPresenceOf(
            properties=#firstName,lastName,email,age,password"
        )>
        <cfset validatesLengthOf(properties=#firstName,lastName," maximum=50)>
        <cfset validatesUniquenessOf(property="email")>
        <cfset validatesNumericalityOf(property="age", onlyInteger=true)>
        <cfset validatesConfirmationOf(property="password")>
    </cffunction>
</cfcomponent>
```

This is fairly readable on its own, but this example defines the following rules that will be run before a create, update, or save is called:

- The `firstName`, `lastName`, `email`, `age`, and `password` fields must be provided, and they can't be blank.

- At maximum, `firstName` and `lastName` can each be up to 50 characters long.
- The value provided for `email` cannot already be used in the database.
- The value for `age` can only be an integer.
- `password` must be provided twice, the second time via a field called `passwordConfirmation`.

If any of these validations fail, Wheels will not commit the create or update to the database. As you'll see later in this chapter, the controller should check for this and react accordingly by showing error messages generated by the model.

Listing of Validation Functions

- `validatesConfirmationOf()`
- `validatesExclusionOf()`
- `validatesFormatOf()`
- `validatesInclusionOf()`
- `validatesLengthOf()`
- `validatesNumericalityOf()`
- `validatesPresenceOf()`
- `validatesUniquenessOf()`
- `validate()`
- `validateOnCreate()`
- `validateOnUpdate()`

Automatic Validations

Now that you have a good understanding of how validations work in the model, here is a piece of good news. By default, Wheels will perform many of these validations for you based on how you have your fields set up in the database.

By default, these validations will run without your needing to set up anything in the model:

- Fields set to `NOT NULL` will automatically trigger `validatesPresenceOf()`.
- Numeric fields will automatically trigger `validatesNumericalityOf()`.
- Date or time fields will be checked for the appropriate format.
- Fields that have a maximum length will automatically trigger `validatesLengthOf()`.

Note these extra behaviors as well:

- Automatic validations will not run for Automatic Time Stamps.
- If you've already set a validation on a particular property in your model, the automatic validations will be overridden by your settings.
- If your database column provides a default value for a given field, Wheels will not enforce a `validatesPresenceOf()` rule on that

property.

To disable automatic validations in your Wheels application, change this setting in `config/settings.cfm`:

```
<cfset set(automaticValidations=false)
```

You can also turn on or off the automatic validations on a per model basis by calling the `automaticValidations()` method from a model's `init()` method.

See the chapter on Configuration and Defaults for more information on available Wheels ORM settings.

Use when, condition, or unless to Limit the Scope of Validation

If you want to limit the scope of the validation, you have 3 arguments at your disposal: `when`, `condition`, and `unless`.

when Argument

The `when` argument accepts 3 possible values.

- `onSave` (the default)
- `onCreate`
- `onUpdate`

To limit our email validation to run only on create, we would change that line to this:

```
<cfset validatesUniquenessOf(property="email", when="onCreate")>
```

condition and unless Arguments

`condition` and `unless` provide even more flexibility when the `when` argument isn't specific enough for your validation's needs.

Each argument accepts a string containing an expression to evaluate. `condition` specifies when the validation should be run. `unless` specifies when the validation should not be run.

As an example, let's say that the model should only verify a CAPTCHA if the user is logged out, but not when they enter their name as "Ben Forta":

```
<cfset validate(  
  method=#validateCaptcha,  
  condition=#not isLoggedIn(),  
  unless=#this.name is 'Ben Forta'  
)>
```

Custom Validations

At the end of the listing above are 3 custom validation functions: `validate()`, `validateOnCreate()`, and `validateOnUpdate()`. These functions allow you to create your own validation rules that aren't covered by Wheels's out-of-the-box functions.

There is only one difference between how the different functions work:

- `validate()` runs on the save event, which happens on both create and update.
- `validateOnCreate()` runs on create.
- `validateOnUpdate()` runs on update.

To use a custom validation, we pass one of these functions a method or set of methods to run:

```
<cfset validate(method=validateEmailFormat)>
```

We then should create a method called `validateEmailFormat`, which in this case would verify that the value set for `this.email` is in the proper format. If not, then the method sets an error message for that field using the `addError()` function.

```
<cffunction name=validateEmailFormat access=private>
  <cfif not IsValid(email, this.email)>
    <cfset addError(property=email, message=Email address is not in a valid format)>
  </cfif>
</cffunction>
```

Note that `IsValid()` is a function build into your CFML engine.

This is a simple rule, but you can surmise that this functionality can be used to do more complex validations as well. It's a great way to isolate complex validation rules into separate methods of your model.

Adding Errors to the Model Object as a Whole

We've mainly focused on adding error messages at a property level, which admittedly is what you'll be doing 80% of the time. But we can also add messages at the model object level with the `addErrorToBase()` function.

As an example, here's a custom validation method that doesn't allow the user to sign up for an account between the hours of 3:00 and 4:00 am in the server's time zone:

```
<cffunction name=disallowMaintenanceWindowRegistrations access=private>
  <cfset var loc = {}>
  <cfset loc.hourNow = DatePart("h", Now())>

  <cfif loc.hourNow gte 3 and loc.hourNow lt 4>
    <cfset loc.timeZone = CreateObject("java", "java.util.TimeZone").getDefault()
    <cfset
      addErrorToBase(
        message=We're sorry, but we don't allow new registrations between
        the hours of 3:00 and 4:00 am #loc.timeZone#.
      )
    >
  </cfif>
</cffunction>
```

Sure, we could add logic to the view to also not show the registration form, but this validation in the model would make sure that data couldn't be posted via a script between those hours as well. Better safe than sorry if you're running a public-facing application!

The Controller

The controller continues with the simplicity of validation setup, and at the most basic level requires only 5 lines of code to persist the form data or return to the original form page to display the list of errors.

```
<cfcomponent extends="Controller" output="false">
    <cffunction name="save">
        <!--- User model from form fields via params --->
        <cfset newUser = model("user").new(params.newUser)

        <!--- Persist new user --->
        <cfif newUser.save()>
            <cfset redirectTo(action="success")>
        <!--- Handle errors --->
        <cfelse>
            <cfset renderPage(action="index")>
        </cfif>
    </cffunction>
</cfcomponent>
```

The first line of the action creates a `newUser` based on the `user` model and the form inputs (via the `params` struct).

Now, to persist the object to the database, the model's `save()` call can be placed within a `<cfif>` test. If the `save` succeeds, the `save()` method will return `true`, and the contents of the `<cfif>` will be executed. But if any of the validations set up in the model fail, the `save()` method returns `false`, and the `<cfelse>` will execute.

The important step here is to recognize that the `<cfelse>` renders the original form input page using the `renderPage()` function. When this happens, the view will use the `newUser` object defined in our `save()` method. If a `redirectTo()` were used instead, the validation information loaded in our `save()` method would be lost.

The View

Wheels factors out much of the error display code that you'll ever need. As you can see by this quick example, it appears to mainly be a normal form. But when there are errors in the provided model, Wheels will apply styles to the erroneous fields.

```
<cfoutput>
#errorMessagesFor("newUser")#

#startFormTag(action="save")#
#textField(label="First Name", objectName="newUser", property="nameFirst")#
#textField(label="Last Name", objectName="newUser", property="nameLast")#
#textField(label="Email", objectName="newUser", property="email")#
#textField(label="Age", objectName="newUser", property="age")#
#passwordField(label="Password", objectName="newUser", property="password")#
#passwordField(
    label="Re-type Password to Confirm", objectName="newUser",
```

```

        property="passwordConfirmation"
    )#
    #submitTag()#
#endFormTag()#
</cfoutput>

```

The biggest thing to note in this example is that a field called `passwordConfirmation` was provided so that the `validatesConfirmationOf()` validation in the model can be properly tested.

For more information on how this code behaves when there is an error, refer to the [Form Helpers and Showing Errors](#) chapter.

Error Messages

For your reference, here are the default error message formats for the different validation functions:

Function	Format
<code>validatesConfirmationOf()</code>	[property] should match confirmation
<code>validatesExclusionOf()</code>	[property] is reserved
<code>validatesFormatOf()</code>	[property] is invalid
<code>validatesInclusionOf()</code>	[property] is not included in the list
<code>validatesLengthOf()</code>	[property] is the wrong length
<code>validatesNumericalityOf()</code>	[property] is not a number
<code>validatesPresenceOf()</code>	[property] can't be empty
<code>validatesUniquenessOf()</code>	[property] has already been taken

Custom Error Messages

Wheels models provide a set of sensible defaults for validation errors. But sometimes you may want to show something different than the default.

There are 2 ways to accomplish this: through global defaults in your config files or on a per-property basis.

Setting Global Defaults for Error Messages

Using basic global defaults for the validation functions, you can set error messages in your config file at `config/settings.cfm`.

```
<cfset set(functionName="validatesPresenceOf," message="Please provide a value for [property]>"
```

As you can see, you can inject the property's name by adding `[property]` to the message string. Wheels will automatically separate words based on your camelCasing of the variable names.

Setting an Error Message for a Specific Model Property

Another way of adding a custom error message is by going into an individual property in the model and adding an argument

named message.

Here's a change that we may apply in the `init()` method of our model:

```
<cfset validatesNumericalityOf(  
  property#email,  
  message#Email address is already in use in another account"  
)>
```

Object Callbacks

Write code that runs every time a given object is created, updated, or deleted.

Callbacks in Wheels allow you to have code executed before and/or after certain operations on an object. This requires some further explanation, so let's go straight to an example of a real-world application: the e-commerce checkout.

A Real-World Example of Using Callbacks

Let's look at a possible scenario for what happens when a visitor to your imaginary e-commerce website submits their credit card details to finalize an order:

- You create a new `order` object using the `new()` method based on the incoming form parameters.
- You call the `save()` method on the `order` object, which will cause Wheels to first validate the object and then store it in the database if it passes validation.
- The next day, you call the `update()` method on the object because the user decided to change the shipping method for the order.
- Another day passes, and you call the `delete()` method on the object because the visitor called in to cancel the order.

Let's say you want to have the following things executed somewhere in the code:

- Stripping out dashes from the credit card number to make it as easy as possible for the user to make a purchase.
- Calculating shipping cost based on the country the package will be sent to.

It's tempting to put this code right in the controller, isn't it? But if you think ahead a little, you'll realize that you might build an administrative interface for orders and maybe an express checkout as well at some point in the future. You don't want to duplicate all your logic in all these places, do you?

Object callbacks to the rescue! By using object callbacks to implement this sort of logic in your model, you keep it out of your controllers and ensure your code staysDRY (Don't Repeat Yourself).

Part of the `Order.cfc` model file:

```
<cfcomponent extends="Model">
  <cffunction name="init">
    <cfset beforeValidationOnCreate="fixCreditCard">
    <cfset afterValidation="calculateShippingCost">
  </cffunction>

  <cffunction name="fixCreditCard">
    Code for stripping out dashes in credit card numbers goes here...
  </cffunction>

  <cffunction name="calculateShippingCost">
    Code for calculating shipping cost goes here...
  </cffunction>
</cfcomponent>
```

■ `</cfcomponent>`

The above code registers 2 methods to be run at specific points in the life cycle of all objects in your application.

Registering and Controlling Callbacks

The following 16 functions can be used to register callbacks.

- `afterNew()` or `afterFind()`
- `afterInitialization()`
- `beforeValidation()`
- `beforeValidationOnCreate()` or `beforeValidationOnUpdate()`
- `afterValidation()`
- `afterValidationOnCreate()` or `afterValidationOnUpdate()`
- `beforeSave()`
- `beforeCreate()` or `beforeUpdate()`
- `afterCreate()` or `afterUpdate()`
- `afterSave()`
- `beforeDelete()`
- `afterDelete()`

Callback Life Cycle

As you can see above, there are a few places (5, to be exact) where one callback or the other will be executed, but not both.

The very first possible callback that can take place in an object's life cycle is either `afterNew()` or `afterFind()`. The `afterNew()` callback methods are triggered when you create the object yourself for the very first time, for example, when using the `new()` method. `afterFind()` is triggered when the object is created as a result of fetching a record from the database, for example, when using `findByKey()`. (There is some special behavior for this callback type that we'll explain in detail later on in this chapter).

The remaining callbacks get executed depending on whether or not we're running a "create," "update," or "delete" operation.

Breaking a Callback Chain

If you want to completely break the callback chain for an object, you can do so by returning `false` from your callback method. (Otherwise, always return `true` or nothing at all.) As an example of breaking the callback chain, let's say you have called the `save()` method on a new object and the method you've registered with the `beforeCreate()` callback returns `false`. As a result, because the method you've registered with the `beforeCreate()` callback will exit the callback chain early by returning `false`, no record will be inserted in the database.

Order of Callbacks

Sometimes you need to run more than one method at a specific point in the object's life cycle. You can do this by passing in a list of method names like this:

```
<cfset beforeSave("checkSomething,checkSomethingElse")>
```

When an object is saved in your application, these two callbacks will be executed in the order that you registered them. The `checkSomething` method will be executed first, and unless it returns `false`, the `checkSomethingElse` method will be executed directly afterward.

Special Case #1: findAll() and the afterFind() Callback

When you read about the `afterFind()` callback above, you may have thought that it must surely only work for `findOne()`/`findByKey()` calls but not for `findAll()` because those calls return query result sets by default, not objects.

Believe it or not, but callbacks are even triggered on `findAll()`! You do need to write your callback code differently though because there will be no `this` scope in the query object. Instead of modifying properties in the `this` scope like you normally would, the properties are passed to the callback method via the `arguments` struct.

Does that sound complicated? This example should clear it up a little. Let's show some code to display how you can handle setting a `fullName` property on a hypothetical `artist` model.

NOTE: Because all `afterFind()` callbacks run when fetching records from the database, it's a good idea to check to make sure that the columns used in the method's logic exist before performing any operations. You mostly encounter this issue when using the `selectargument` on a finder to limit the number of column returned. But no worries! You can use `StructKeyExists()` and perform a simple check to make sure that the columns exists in the `arguments` scope.

```
<cfcomponent extends="Model" output="false">
    <cffunction name="init">
        <cfset afterFind("setFullName")>
    </cffunction>

    <cffunction name="setFullName">
        <cfset arguments.fullName = ">
        <cfif
            StructKeyExists(argument$firstName)
            and StructKeyExists(argument$lastName)
        >
            <cfset arguments.fullName = Trim(
                arguments.firstName"&" & arguments.lastName
            )
        </cfif>
        <cfreturn arguments>
    </cffunction>
</cfcomponent>
```

In our example model, an artist's name can consist of both a first name and a last name ("John Mayer") or just the band / last name ("Abba."). The `setFullName()` method handles the logic to concatenate the names.

Always remember to return the `arguments` struct, otherwise Wheels won't be able to tell that you actually wanted to make any changes to the query.

Special Case # 2: Callbacks and the `updateAll()` and `deleteAll()` Methods

Please note that if you use the `updateAll()` or the `deleteAll()` methods in Wheels, they will not instantiate objects by default, and thus any callbacks will be skipped. This is good for performance reasons because if you update 1,000 records at once, you probably don't want to run the callbacks on each object. Especially not if they involve database calls.

However, if you want to execute all callbacks in those methods as well, all you have to do is pass in `instantiate=true` to the `updateAll()/ deleteAll()` methods.

Calculated Properties

Generate extra properties in your models on the fly without needing to store redundant data in your database.

Working within CFML's Constraints to Deliver OOP-like Functionality

Wheels makes up for the slowness of arrays of objects in CFML by providing *calculated properties*. With calculated properties, you can generate additional properties on the fly based on logic and data within your database.

Example #1: Full Name

Consider the example of `fullName`. If your database table has fields for `firstName` and `lastName`, it wouldn't make sense to store a third column called `fullName`. This would require more storage for redundant data, and it would add extra complexity that could lead to bugs and maintenance problems in the future.

Traditional Object-Oriented Calculations

In most object-oriented languages, you would add a method to your class called `getFullName()`, which would return the concatenation of `this.firstName` & " " & `this.lastName`. The `getFullName()` method could potentially provide arguments to list the last name first and other types of calculations or transformations as well.

Wheels still allows for you to do this sort of dynamic calculation with the `returnAs="objects"` argument in methods like `findAll()`, but we advise against it when fetching large data sets because of the slowness of `CreateObject()` across CFML engines.

See the chapter on Reading Records for more information.

Using Calculated Properties to Generate `fullName` in the Database at Runtime

As an alternative, you can set up a calculated property that dynamically performs the concatenation at the database level. In our example, we would write a line similar to this in our model's `init()` method:

```
<cfset
  property(
    name=#fullName",
    sql=#RTRIM(LTRIM(ISNULL(users.firstname, '')) + ' '
      + ISNULL(users.lastname, ''))"
  )
>
```

As you can probably deduce, we're creating a SQL statement that will be run in the `SELECT` clause to generate the `fullName`.

With this line in place, `fullName` will become available in both full model objects and query objects returned by the various finder methods like `findAll()` and `findOne()`.

Example #2: Age

Naturally, if you store the user's birth date in the database, your application can use that data to dynamically calculate the user's age. Your app always knows how many years old the user is without needing to explicitly store his or her age.

Creating the Calculated Property for Age

In order to calculate an extra property called `age` based on the `birthDate` column, our calculated property in `init()` may look something like this:

```
<cfset
  property(
    name=#age",
    sql#(CAST(CONVERT(CHAR(8), GETDATE(), 112) AS INT)
        - CAST(CONVERT(CHAR(8), users.date_of_birth, 112) AS INT))
        / 10000"
  )
>
```

Much like the `fullName` example above, this will cause the database to add a property called `age` storing the user's age as an integer.

Note that the cost to this approach is that you may need to introduce DBMS-specific code into your models. This may cause problems when you need to switch DBMS platforms, but at least all of this logic is isolated into your model CFCs.

Using the New `age` Property for Other Database Calculations

Calculated properties don't end at just generating extra properties. You can now also use the new property for additional calculations:

- Creating additional properties with the `select` argument
- Additional `where` clause calculations
- Record sorting with `order`
- Pagination
- And so on...

For example, let's say that we only want to use `age` to return users who are in their 20s. We can use the new `age` property as if it existed in the database table. For extra measure, let's also sort the results from oldest to youngest.

```
<cfset
  users = model("user").findAll(
    where=#age >= 20 AND age < 30," order=#age DESC"
```

|| >)

Transactions

Wheels automatically wraps your database calls in transactions to assist your application in maintaining data integrity. Learn how to control this functionality.

Database transactions are a way of grouping multiple queries together. They are useful in case the outcome of one query depends on the completion of another. For example, if you want to take money from one person's bank account, and transfer it into someone else's, you probably want to make sure the debit completes before running the credit.

You'll be pleased to know that Wheels makes using database transactions easy. In fact, the vast majority of the time, you won't need to think about using them at all because Wheels automatically runs all queries within the callback chain as a single transaction for creates, updates, and deletes.

If any of the callbacks within the chain return false, none of the queries will commit.

For example, say you want to automatically create the first post when a new author subscribes to a blog.

In your `blog` model, you would add the following code:

```
<!-- Author.cfc Model -->
<cffunction name="init">
  <cfset afterCreate("createFirstPost")>
</cffunction>

<cffunction name="createFirstPost">
  <cfset var post = model("post").new(
    authorId=this.id,
    text="This is my first post!"
  )>
  <cfreturn post.save()>
</cffunction>
```

In this example, if the post doesn't save (perhaps due to a validation problem), the author doesn't get created either. This helps to maintain database integrity.

Disabling Transactions

If you want to manage transactions yourself using the `<cftransaction>` tag, you can simply add `transaction="none"` to any CRUD method.

```
<cfset model("author").create(name="John", transaction="none")>
```

Another option is to disable transactions across your entire application using the `transactionMode` configuration:

```
<!-- In `config/settings.cfm` -->
<cfset set(transactionMode="none")>
```

See the chapter about Configuration and Defaults for more details.

Using Rollbacks

Sometimes it's useful to use a rollback to test a process without making any permanent changes to the database. To do this, add `transaction="rollback"` to any CRUD method.

```
<cfset model("author").create(name#John", transaction#rollback)>
```

Again, to configure your entire application to rollback *all* transactions, you can set the `transactionMode` configuration to `rollback`.

```
<!--- In `config/settings.cfm` --->  
<cfset set(transactionMode#rollback)>
```

Nesting Transactions with `invokeWithTransaction`

One issue with ColdFusion is that you cannot nest `<cftransaction>` tags. In this case, Wheels provides a workaround. If you wish to run a method within a transaction, use `invokeWithTransaction()`, as below.

```
<cfset invokeWithTransaction(  
    method#transferFunds"  
    personFrom=david,  
    personTo=mary,  
    amount=100  
)>
```

Dirty Records

How to track changes to objects in your application.

Wheels provides some very useful methods for tracking changes to objects. You might think, *Why do I need that? Won't I just know that I changed the object myself?*

Well, that depends on the structure of your code.

As you work with Wheels and move away from that procedural spaghetti mess you used to call code to a better, cleaner object-oriented approach, you may get a sense that you have lost control of what your code is doing. Your new code is creating objects, they in turn call methods on other objects automatically, methods are being called from multiple places, and so on. Don't worry though, this is a good thing. It just takes a while to get used to, and with the help of some Wheels functionality, it won't take you that long to get used to it either.

An Example with Callbacks

One area where this sense of losing control is especially noticeable is when you are using *callbacks* on objects (see the chapter on Object Callbacks for more info). So let's use that for our example.

Let's say you have used a callback to specify that a method should be called whenever a `user` object is saved to the database. You won't know exactly **where** this method was called from. It could have been the user doing it themselves on the website, or it could have been done from your internal administration area. Generally speaking, you don't need to know this either.

One thing your business logic might need to know though is a way to tell exactly **what** was changed on the object. Maybe you want to handle things differently if the user's last name was changed than if the email address was changed, for example.

Let's look at the methods Wheels provide to make tracking these changes easier for you.

Methods for Tracking Changes

Let's get to coding...

```
<cfset post = model{post}.findByKey(1)>  
<cfset result = post.hasChanged*>
```

Here we are using the `hasChanged()` method to see if any of the object properties has changed.

By the way, when we are talking about "change" in Wheels, we always mean whether or not an object's properties have changed compared to what is stored in the columns they map to in the database table.

In the case of the above example, the `result` variable will contain `false` because we just fetched the object from the database and did not make any changes to it at all.

Well, let's make a change then. If we didn't, this chapter wouldn't be all that interesting, would it?

```
<cfset post.title = "A New Post Title"
<cfset result = post.hasChanged()
```

Now `result` will be `true` because what is stored in `post.title` differs from what is stored in the `title` column for this record in the `posts` table (well, unless the title was "A New Post Title" even before the change, in which case the result would still be `false`).

When calling `hasChanged()` with no arguments, `Wheels` will check **all** properties on the object and return `true` if any of them have changed. If you want to see if a specific property has changed, you can pass in `property="title"` to it or use the dynamic method `XXXHasChanged()`. Replace `XXX` with the name of the property. In our case, the method would then be named `titleHasChanged()`.

If you want to see what a value was before a change was made, you can do so by calling `changedFrom()` and passing in the name of a property. This can also be done with the dynamic `XXXChangedFrom()` method.

When an object is in a changed state, there are a couple of methods you can use to report back on these changes. `changedProperties()` will give you a list of the property names that have been changed. `allChanges()` returns a struct containing all the changes (both the property names and the changed values themselves).

If you have made changes to an object and for some reason you want to revert it back, you can do so by calling `reload()` on it. This will query the database and update the object properties with their corresponding values from the database.

OK, let's save the object to the database now and see how that affects things.

```
<cfset post.save()
<cfset result = post.hasChanged()
```

Now `result` will once again contain `false`. When you save a changed (a.k.a. "dirty") object, it clears out its changed state tracking and is considered unchanged again.

Don't Forget the Context

All of the examples in this chapter look a little ridiculous because it doesn't make much sense to check the status of an object when you changed it manually in your code. As we said in the beginning of the chapter, when put into context of callbacks, multiple methods, etc., it will become clear how useful these methods really are.

Internal Use of Change Tracking

It's worth noting here that `Wheels` makes good use of this change tracking internally as well. If you make changes to an object, `Wheels` is smart enough to only update the changed columns, leaving the rest alone. This is good for a number of reasons but perhaps most importantly for database performance. In high traffic web applications, the bottleneck is often the database, and anything that can be done to prevent unnecessary database access is a good thing.

One "Gotcha" About Tracking Changes

If you create a brand new object with the `new()` method and call `hasChanged()` on it, it will return `true`. The reason for this seemingly unexpected behavior is that change is always viewed from the database's perspective. The `hasChanged()` method will return `true` in this case because it is different from what is stored in the database (i.e. it doesn't exist at all in the database yet).

If you would simply like to know if an object exists in the database or not, you can use the `isNew()` method.

Soft Delete

An easy way to keep deleted data in your database.

"Soft delete" in database lingo means that you set a flag on an existing table which indicates that a record has been deleted, instead of actually deleting the record.

How to Use Soft Deletion

If you create a new date column (the column type will depend on your database vendor, but usually you want to use `date`, `datetime`, or `timestamp`) on a table and name it `deletedAt`, Wheels will automatically start using it to record soft deletes.

Without the soft delete in place, a `delete()` call on an object will delete the record from the table using a `DELETE` statement. With the soft delete in place, an `UPDATE` statement is sent instead (that sets the `deletedAt` field to the current time).

Of course, all other Wheels functions are smart enough to respect this. So if you use a `findAll()` function, for example, it will not return any record that has a value set in the `deletedAt` field.

What this all means is that you're given a convenient way to keep deleted data in your database forever, while having your application function as if the data is not there.

Obviously, if you have any manual queries in your application, you'll need to remember to add `deletedAt IS NULL` to the `WHERE` part of your `SQL` statements.

Automatic Time Stamps

Let Wheels handle time stamping of records.

When working with database tables, it is very common to have a column that holds the time that the record was added or last modified. If you have an e-commerce website with an orders table, you want to store the date and time the order was made; if you run a blog, you want to know when someone left a comment; and so on.

As with anything that is a common task performed by many developers, it makes a good candidate for abstracting to the framework level. So that's what we did.

Columns Used for Timestamps

If you have either of the following columns in your database table, Wheels will see them and treat them a little differently than others.

`createdAt`

Wheels will use a `createdAt` column automatically to store the current date and time when an `INSERT` operation is made (which could happen through a `save()` or `create()` operation, for example).

`updatedAt`

If Wheels sees an `updatedAt` column, it will use it to store the current date and time automatically when an `UPDATE` operation is made (which could happen through a `save()` or `update()` operation, for example).

Data Type of Columns

If you add any of these columns to your table, make sure they can accept date/time values (like `datetime` or `timestamp`, for example) and that they can be set to `null`.

Using Multiple Data Sources

How to use more than one database in your Wheels application.

Sometimes you need to pull data from more than one database, whether it's by choice (for performance or security reasons, perhaps) or because that's the way your infrastructure is set up. It's something you have to find a way to deal with.

Wheels has built-in functionality for this so that you don't have to revert back to writing the queries and setting the data source manually whenever you need to use a data source other than the default one. In order accomplish this, you will use the `dataSource()` function.

Using the `dataSource()` Function

Overriding the default data source is done on a per model basis in Wheels by calling the `dataSource()` function from within your model's `init()` method. By doing this, you instruct wheels to use that data source whenever it interacts with that model.

Here's an example of a model file:

```
<cfcomponent extends="Model">
  <cffunction name="init">
    <cfset dataSource(mySecondDatabase)>
  </cffunction>
</cfcomponent>
```

It's important to note that in order for Wheels to use the data source, it must first be configured in your respective CFML engine (i.e. in the Adobe ColdFusion or Railo Administrator).

Does Not Work with Associations

One thing to keep in mind when using multiple data sources with Wheels is that it doesn't work across associations. When including another model within a query, Wheels will use the calling model's data source for the context of the query.

Let's say you have the following models set up:

models/Photo.cfc:

```
<cfcomponent extends="Model">
  <cffunction name="init">
    <cfset dataSource(myFirstDatabase)>
    <cfset hasMany("photoGalleries")>
  </cffunction>
</cfcomponent>
```

models/PhotoGallery.cfc:

```
<cfcomponent extends="Model">
    <cffunction name="init">
        <cfset dataSource="mySecondDatabase">
    </cffunction>
</cfcomponent>
```

Because the `photo` model is the main model being used in the following example, its data source (`myFirstDatabase`) will be the one used in the query that `findAll()` ends up executing.

```
<cfset myPhotos = model("photo").findAll(include="photoGalleries">
```

Pages

Where to place your view files and what to put in them.

We've talked previously about how the controller is responsible for deciding which view files to render to the user. Read the Rendering Content chapter if you need to refresh your memory about that topic.

In this chapter, we'll explain exactly *where* to place these files and *what* to put in them.

Where to Place the Files

In the simplest case, your controller action (typically a function inside your controller CFC file) will have a view file associated with it. As explained in the Rendering Content chapter, this file will be included automatically at the end of the controller action code. So if you're running the `show` action in the `blog` controller, for example, Wheels will include the `views/blog/show.cfm` file.

Some rules can be spotted here:

- All view files live in the `views` folder.
- Each controller gets a subfolder named after it in the `views` folder.
- The view file to include is just a regular `.cfm` file named after the action.

For creating standard pages, your work process will likely consist of the following steps:

1. Create the controller action (a function in the controller CFC file).
2. Create the corresponding view file for it (a `.cfm` file in the controller's view folder).

There can be some exceptions to this process though, so let's go through some possible scenarios.

Controller Actions Without Associated View Files

Not all controller actions need a corresponding view file. Consider the case where you process a form submission. To make sure it's not possible for the user to refresh the page and cause multiple submissions, you may choose to perform the form processing and then send the user directly to another page using the `redirectTo()` function.

Rendering the View File for Another Action

Sometimes you want the controller action to render the view file for a different action than the one currently executing. This is especially common when your application processes a form and the user makes an input error. In this case, you'll probably choose to have your application display the same form again for correction.

In this case, you can use the `renderPage()` function and specify a different action in the `action` argument (which will include the

view page for that action but **not** run the controller code for it).

Sharing a View File Between Actions

Sometimes it's useful to have a view file that can be called from several controller actions. For these cases, you'll typically call `renderPage()` with the `template` argument.

When using the `template` argument, there are specific rules that Wheels will follow in order to locate the file you want to include:

- If the `template` argument starts with the `/` character, Wheels will start searching from the root of the `views` folder. Example: `renderPage(template="/common/page")` will include the `views/common/page.cfm` file.
- If it contains the `/` character elsewhere in the string, the search will start from the controller's view folder. Example: `renderPage(template="misc/page")` will include the `views/blog/misc/page.cfm` file if we're currently in the `blog` controller.
- In all other cases (i.e. when the `template` argument does not contain the `/` character at all), Wheels will just assume the file is in the controller's view folder and try to include it. Example: `renderPage(template="something")` will include the `views/blog/something.cfm` file if we're currently in the `blog` controller.

Also note that both `renderPage(template="thepage")` and `renderPage(template="thepage.cfm")` work fine. But most of the time, Wheels developers will tend to leave out the `.cfm` part.

What Goes in the Files?

This is the output of your application: what the users will see in their browsers. Most often this will consist of HTML, but it can also be JavaScript, CSS, XML, etc. You are of course free to use any CFML tags and functions that you want to in the file as well. (This is a CFML application, right?)

In addition to this normal code that you'll see in most ColdFusion applications—whether they are made for a framework or not—Wheels also gives you some nice constructs to help keep your code clean. The most important ones of these are `Layouts`, `Partials`, and `Helpers`.

When writing your view code, you will have access to the variables you have set up in the controller file. The idea is that the variables you want to access in the view should be set unscoped (or in the `variables` scope if you prefer to set it explicitly) in the controller so that they are available to the view template.

In addition to the variables you have set yourself, you can also access the `params` struct. This contains anything passed in through the URL or with a form. If you want to follow MVC rules more closely though, we recommend only accessing the `params` struct in the controller and then setting new variables for the information you need access to in the view.

The most important thing to remember when creating your view is to be careful not to put too much code in there. Avoid code dealing with the incoming request (this can be moved to the controller) and code containing business logic (consider moving this to a model). If you have view-related code but too much of it, it may be beneficial to break it out into a helper or a partial.

Cleaning Up Output

A view's job is also to clean up and format the values provided by the controller before being displayed. This is especially important when content from a data source is not HTML-escaped.

For example, if the view is to display the `title` column from a query object called `posts`, it should escape HTML special characters:

```
<ul>  
  <cfoutput query="posts">  
    <li>#HtmlEditFormat(posts.title)</li>  
  </cfoutput>  
</ul>
```

Partials

Simplify your views by breaking them down into partial page templates.

Partials in Wheels act as a wrapper around the good old `<cfinclude>` tag. By calling `includePartial()` or `renderPartial()`, you can include other view files in a page, just like `<cfinclude>` would. But at the same time, partials make use of common Wheels features like layouts, caching, model objects, and so on.

These functions also add a few cool things to your development arsenal like the ability to pass in a query or array of objects and have the partial file called on each iteration (to name one).

Why Use a Partial?

Websites often display the same thing on multiple pages. It could be an advertisement area that should be displayed in an entire section of a website or a shopping cart that is displayed while browsing products in a shop. You get the idea. To avoid duplicating code, you can place it in a file (the "partial" in Wheels terms) and include that file using `includePartial()` on the pages that need it.

Even when there is no risk of code duplication, it can still make sense to use a partial. Breaking up a large page into smaller, more manageable chunks will help you focus on each part individually.

If you've used `<cfinclude>` a lot in the past (and who hasn't?!), you probably already knew all of this though, right?

Storing Your Partial Files

To make it clear that a file is a partial and not a full page, we start the filename with an underscore character. You can place the partial file anywhere in the `views` folder. When locating partials, Wheels will use the same rules as it does for the `template` argument to `renderPage()`. This means that if you save the partial in the current controller's view folder, you reference it simply by its name.

For example, if you wanted to have a partial for a comment in your `blog` controller, you would save the file at `views/blog/_comment.cfm` and reference it (in `includePartial()` and `renderPartial()`) with just "comment" as the first argument.

Sometimes it's useful to share partials between controllers though. Perhaps you have a banner ad that should be displayed across several controllers. One common approach then is to save them in a dedicated folder for this at the root of the `views` folder. To reference partials in this folder, in this case named `shared`, you would then pass in `"/shared/banner"` to `includePartial()` instead.

Making the Call

Now that we know why we should use partials and where to store them, let's make a call to `includePartial()` from a view page to have Wheels display a partial's output.

```
■ <cfoutput#includePartial("banner"*)#cfoutput>
```

That code will look for a file named `_banner.cfm` in the current controller's view folder and include it.

Let's say we're in the `blog` controller. Then the file that will be included is `views/blog/_banner.cfm`.

As you can see, you don't need to specify the `.cfm` part or the underscore when referencing a partial.

Passing in Data

You can pass in data by adding named arguments on the `includePartial()` call. Because we use the `partial` argument to determine what file to include, you can't pass in a variable named `partial` though. The same goes for the other arguments as well, like `layout`, `spacer`, and `cache`.

Here is an example of passing in a title to a partial for a form:

```
■ <cfoutput>
  #includePartial(partial="loginRegisterForm", title="Please log in here")#
</cfoutput>
```

Now you can reference the title variable as `arguments.title` inside the `_loginregisterform.cfm` file.

If you prefer, you can still access the view variables that are set outside of the partial. The advantage with specifically passing them in instead is that they are then scoped in the `arguments` struct (which means less chance of strange bugs occurring due to variable conflicts). It also makes for more readable and maintainable code. (You can see the intent of the partial better when you see what is passed in to it).

Automatic Calls to a Data Function

There is an even more elegant way of passing data to a partial though. When you start using a partial on several pages on a site spread across multiple controllers, it can get quite cumbersome to remember to first load the data in an appropriate function in the controller, setup a before filter for it, pass that data in to the partial, and so on.

Wheels can automate this process for you. The convention is that a partial will always check if a function exists on the controller with the same name as the partial itself (and that it's set to `private` and will return a struct). If so, the partial will call the function and add the output returned to its `arguments` struct.

This way, the partial can be called from anywhere and acts more like a "black box." All communication with the model is kept in the controller as it should be for example.

If you don't want to load the data from a function with the same name as the partial (perhaps due to it clashing with another function name), you can specify the function to load data from with the `dataFunction` argument to `includePartial()` and `renderPartial()`.

Partials with Layouts

Just like a regular page, Wheels partials also understand the concept of layouts. To use this feature, simply pass in the name of the layout file you want to wrap the partial in with the `layout` argument, like this:

```
<cfoutput>
#includePartial(partial="newsItem", layout="/boxes/blue")#
</cfoutput>
```

This will wrap the partial with the code found in `views/boxes/_blue.cfm`. Just like with other layouts, you use `includeContent()` to represent the partial's content.

That said, your `_blue.cfm` file could end up looking something like this:

```
<div class=#news">
  <cfoutput#includeContent()#>
</div>
```

One difference from page layouts is that the layout file for partials has to start with the underscore character.

It's also worth noting that it's perfectly acceptable to include partials inside layout files as well. This opens up the possibility to nest layouts in complex ways.

Caching a Partial

Caching a partial is done the same way as caching a page. Pass in the number of minutes you want to cache the partial for to the `cache` argument.

Here's an example where we cache a partial for 15 minutes:

```
<cfoutput>
#includePartial(partial="userListing", cache=15)#
</cfoutput>
```

Using Partial with an Object

Because it's quite common to use partials in conjunction with objects and queries, Wheels has built-in support for this too. Have a look at the code below, which passes in an object to a partial:

```
<cfset cust = model(customer).findByKey(params.key)>
<cfoutput#includePartial(cust)#>
```

That code will figure out that the `cust` variable contains a `customer` model object. It will then try to include a partial named `_customer.cfm` and pass in the object's properties as arguments to the partial. There will also be an `object` variable available in the `arguments` struct if you prefer to reference the object directly.

Try that code and `<cfdump>` the `arguments` struct inside the partial file, and you'll see what's going on. Pretty cool stuff, huh?

Using Partials with a Query

Similar to passing in an object, you can also pass in a query result set to `includePartial()`. Here's how that looks:

```
<cfset customers = model("customer").findAll()>
<cfoutput>#includePartial(customers)#</cfoutput>
```

In this case, Wheels will iterate through the query and call the `_customer.cfm` partial on each iteration. Similar to the example with the object above, Wheels will pass in the objects' properties (in this case represented by records in the query) to the partial.

In addition to that, you will also see that a counter variable is available. It's named `current` and is available when passing in queries or arrays of objects to a partial.

The way partials handle objects and queries makes it possible to use the exact same code inside the partial regardless of whether we're dealing with an object or query record at the time.

If you need to display some HTML in between each iteration (maybe each iteration should be a list item for example), then you can make use of the `spacer` argument. Anything passed in to that will be inserted between iterations. Here's an example:

```
<cfoutput>
<ul>
  <li>#includePartial(partial=customers, spacer="/li><li>")#</li>
</ul>
</cfoutput>
```

Partials and Grouping

There is a feature of CFML that is very handy: the ability to output a query with the `group` attribute. Here's an example of how this can be done with a query that contains artists and albums (with the artist potentially being duplicated since they can have more than one album):

```
<cfoutput query=artistsAndAlbums "group=artistid">
  <!--- Artist info is displayed just once for each artist here --->
  <cfoutput>
    <!--- Each album is looped here --->
  </cfoutput>
</cfoutput>
```

We have ported this great functionality into calling partials with queries as well. Here's how you can achieve it:

```
#includePartial(partial=artistsAndAlbums, group="artistId")#
```

When inside the partial file, you'll have an additional subquery made available to you named `group`, which contains the albums for

the current artist in the loop.

Using Partial with an Array of Objects

As we've hinted previously in this chapter, it's also possible to pass in an array of objects to a partial. It works very similar to passing in a query in that the partial is called on each iteration.

Rendering or Including?

So far we've only talked about `includePartial()`, which is what you use from within your views to include other files. There is another similar function as well: `renderPartial()`. This one is used from your controller files when you want to render a partial instead of a full page. At first glance, this might not make much sense to do. There is one common usage of this though—AJAX requests.

Let's say that you want to submit comments on your blog using AJAX. For example, the user will see all comments, enter their comment, submit it, and the comment will show up below the existing ones without a new page being loaded.

In this case, it's useful to use a partial to display each comment (using `includePartial()` as outlined above) and use the same partial when rendering the result of the AJAX request.

Here's what your controller action that receives the AJAX form submission would look like:

```
<cfset comment = model("comment").create(params.newComment)
<cfset renderPartial(comment)
```

Please note that there currently is no support for creating the AJAX form directly with Wheels. This can easily be implemented using a JavaScript library such as jQuery or Prototype though.

Linking Pages

Wheels does some magic to help you link to other pages within your app. Read on to learn why you'll rarely use an `<a>` tag ever again.

Wheels's built-in `linkTo()` function does all of the heavy lifting involved with linking the different pages of your application together. You'll generally be using `linkTo()` within your view code.

As you'll soon realize, the `linkTo()` function accepts a whole bunch of arguments. We won't go over all of them here so don't forget to have a look at the documentation for `linkTo()` for the complete details.

Let's go right to an example...

Basic Example

If we wanted to link to the page that is the `authors` action in the `blog` controller, this is the code that we would write:

```
#linkTo(text="List of Authors", controller="blog", action="authors")#
```

The above code would generate this markup:

```
<a href="/blog/authors">List All Authors</a>
```

Extreme Example

If we were to use all of the parameters except for `route` (more on that later), our code may look something like this:

```
#linkTo(
  text="Wheels Rocks!", controller="wheels", action="rocks", key=55,
  params="rocks=yes&referral=cfwheels.com", anchor="rockin",
  host="www.securesite.com", protocol="https", onlyPath=false,
  confirm="Are you sure that Wheels rocks?"
)#
```

Which would generate this HTML:

```
<a
  href="https://www.securesite.com/wheels/rocks/55?rocks=yes&referral=cfwheels.com#rockin"
  onclick="return confirm('Are you sure that Wheels rocks?')">Wheels Rocks</a>
```

Using `linkTo()` with Routes

Routes makes creating custom URLs outside of the wheels `controller/action/key` convention pretty darn easy. Refer to the [Using Routes](#) chapter for more about how to setup your routes.

Let's say that you have this route set up in your `config/routes.cfm` file:

```
<cfset addRoute(
  name=#profile",
  pattern=#user/[username]",
  controller=#account", action=#profile"
)>
```

You can then pass the route name as the route argument of `linkTo()`. With this, you can also pass the `username` parameter that the `profile` route requires like so. (Pretend that we have instantiated a `user` object that has the fields `firstName`, `lastName`, and `username`.)

```
#linkTo(text="#user.firstName# #user.lastName#", route="profile", username=user.username)#
```

Images as Links

If you'd like to use an image as a link to another page, simply pass the output of `imageTag()` to the `text` argument of `linkTo()`:

```
#linkTo(text="imageTag(source="authors.jpg")", controller="blog", action="authors")#
```

Adding Additional Attributes Like `class`, `rel`, and `id`

Like many of the other Wheels view helpers, any additional arguments that you pass to `linkTo()` will be added to the generated `<a>` tag as attributes.

For example, if you'd like to add a `class` attribute value of `button` to your link, here's what the call to `linkTo()` would look like:

```
#linkTo(text="Add to Cart", controller="cart", action="add", class="button")#
```

The same goes for any other argument that you pass, including but not limited to `id`, `rel`, `onclick`, etc.

What If I Don't Have URL Rewriting Enabled?

Wheels will handle linking to pages without URL rewriting for you automatically. Let's pretend that you still have Wheels installed in your site root, but you do not have URL rewriting on. How you write your `linkTo()` call will not change:

```
#linkTo(text="This link still works", controller="company", action="contact", key=3)#
```

But Wheels will still correctly build the link markup:

```
■ <a href="/index.cfm/company/contact/">This link still works</a>
```

Linking in a Subfolder Deployment of Wheels

The same would be true if you had Wheels installed in a subfolder, thus perhaps eliminating your ability to use URL Rewriting (depending on what web server you have). The same `linkTo()` code above may generate this HTML if you had Wheels installed in a subfolder called `foo`:

```
■ <a href="/foo/index.cfm?controller=company&action=contact&key=3"
  This link still works>
```

Use the `linkTo()` Function for Portability

An `<a>` tag is easy enough, isn't it? Why would we need to use a function to do this mundane task? It turns out that there are some advantages. Here's the deal.

Wheels gives you a good amount of structure for your applications. With this, instead of thinking of URLs in the "old way," we think in terms of what route, controller, and/or action we're sending the user to next.

What's more, Wheels is smart enough to build URLs for you. And it'll do this for you based on your situation with URL rewriting. Are you using URL rewriting in your app? Great. Wheels will build your URLs accordingly. Not fortunate enough to have URL rewriting capabilities in your development or production environments? That's fine too because Wheels will handle that automatically. Are you using Wheels in a subfolder on your site, thus eliminating your ability to use URL rewriting? Wheels handles that for you too.

If you see the pattern, this gives your application a good deal of portability. For example, you could later enable URL rewriting or move your application to a different subfolder. As long as you're using `linkTo()` to build your links, you won't need to change anything extra to your code in order to accommodate this change.

Oh, and another reason is that it's just plain cool too. ;)

Using Layouts

Wheels allows you to create layouts so that you don't need to `<cfinclude>` header and footer code on every single view template. We'll show you how to setup default layouts, controller- and action-specific layouts, and layouts for your emails.

Introduction

As a red-blooded CFML developer, you're used to creating include files like `header.cfm` and `footer.cfm`, and then using `<cfinclude>` on every single page to include them. The popular way to do it looks something like this:

```
<cfinclude template=/includes/header.cfm">
<p>Some page content</p>
<cfinclude template=/includes/footer.cfm">
```

Does that mean that you should `<cfinclude>` your headers and footers in every view in your Wheels app? Heck no! If the structure of your pages ever changed, you would need to edit *every single page* on your site to make the fix.

Layouts to the rescue!

Implementing a Layout

In your Wheels installation, layout files are stored either directly in the root of the `views` folder or contained in one of the controllers' view folders. Let's go over how layouts work, starting with the simplest way and then moving on to more complex setups.

Let's say that you want to define one layout to be used by every view in your application. You would accomplish this by editing the *default layout*. The default layout is the `views/layout.cfm` file. In a fresh install of Wheels, you'll notice that it only contains a few lines of code:

```
<html>
  <body>
    <cfoutput#includeContent()#</cfoutput>
  </body>
</html>
```

The call to `includeContent()` represents the output of your page generated by your view files. Whatever code you put before this snippet will be run before the view. Similarly, whatever code you put after the snippet will be run afterward.

Simple Example

For most purposes, this means that you could write code for your page header before the snippet, and write code for the footer after. Here is a simple example of wrapping your view's content with a header and footer.

```
<html>
<head>
<title><cfoutput#title#/cfoutput>/title>
</head>

<body>

<div id=#container#>
  <div id=#navigation#>
    <ul>
      <cfoutput>
      <li>#linkTo(text="Home", controller="main")#</li>
      <li>#linkTo(text="About Us", controller="about")#</li>
      <li>#linkTo(text="Contact Us", controller="contact")#</li>
      </cfoutput>
    </ul>
  </div>
  <div id=#content#>
    <cfoutput#includeContent()#/cfoutput>
  </div>
</div>

</body>
</html>
```

As you can see, we just wrote code that wraps every view's content with the layout. Pretty cool, huh?

Use of Variables in the Layout

Just like views in Wheels, any variable declared by your application's controller can be used within your layouts. In addition to that, any variables that you set in view templates are accessible to the layouts as well.

Notice in the above code example that there is a variable called `title` being output in between the `<title>` tags. This would require that any controller or view using this particular layout would need to set a variable named `title`.

To help document this, you can use `<cfparam>` tags at the top of your layout files. That way any developer using your layout in the future could easily see which variables need to be set by the controller.

Here's an example:

```
<!-- Title is required -->
<cfparam name=#title# type=#string#>

<cfoutput>

<html>
<head>
<title>#title#/title>
</head>
```

```
<body>
<!-- View's Content --->
<h1>#title#/h1>
#contentForLayout()#
</body>
</html>

</cfoutput>
```

There's also a different way to set variables that goes hand in hand with the `includeContent()` function that you may prefer. It's the `contentFor()` function. We'll dig into how that one works later in this chapter.

The Default Layout

One layout file that is already created for you is `views/layout.cfm`. Think of it as the default layout to be used by any given controller.

If you're writing code for a controller called `press` and there is no layout created for `press`, Wheels will automatically use `views/layout.cfm` as the layout.

If you implement a layout for the `press` controller, then that layout will be used instead of the default layout.

So, how exactly do you implement a layout meant specifically for just one controller? Well, that's next...

Overriding the Default Layout with a Controller-Specific Layout

Let's pretend that you want to create a layout to be used only in a controller called `blog`. To do this, you would simply create the layout and save it as `views/blog/layout.cfm`.

As you can see, the convention is to place your layout file together with the other view files for the controller.

Overriding the Default Layout Using the `usesLayout()` Function

However, if you need to override the name of the layout file or its location in the folder structure, you can specify what layout file to use with the `usesLayout()` function in the controller's `init` function instead.

```
<cffunction name="init">
  <cfset usesLayout('blogLayoutOne')>
</cffunction>
```

With that code placed in the `controllers/Blog.cfc` file, all actions will now wrap their contents with the `bloglayoutone.cfm` file instead of the `layout.cfm` file.

The `usesLayout()` function also accepts `except`, `only`, and `useDefault` arguments for further customization.

```
<cffunction name="init">
    <cfset usesLayout(name="blogLayoutOne" except="home")>
</cffunction>
```

That code tells Wheels to apply the `blogLayoutOne` layout for any actions in this controller except for the `home` action. In the case of the `home` action, it will fall back to the default behavior (i.e. using the `views/blog/layout.cfm` file).

If for some reason you do not want the default behavior to be used when conditions aren't met, you can set the `useDefault` argument to `false`.

```
<cffunction name="init">
    <cfset usesLayout(name="blogLayoutOne" except="home", useDefault=false)>
</cffunction>
```

You can even instruct Wheels to run a specific function that will determine the layout handling.

```
<cffunction name="init">
    <cfset usesLayout("resolveLayout")>
</cffunction>

<cffunction name="resolveLayout">
    <cfswitch expression="chapterPdf">
        <cfcase value="index">
            <cfreturn "index_layout">
        </cfcase>
        <cfcase value="show">
            <cfreturn "show_layout">
        </cfcase>
    </cswitch>
</cffunction>
```

It's worth repeating here that everything inside the controller's `init()` function runs only once per application and controller. This is why you can't perform the logic that decides which layout to use directly inside the `init()` function itself. Instead, you tell Wheels to always run the `resolveLayout` function on each incoming request.

Overriding the Default Layout at the Action Level

Another option for overriding layouts is to use the `layout` argument of the `renderPage()` function.

As you may already know, Wheels's `renderPage()` function is the last thing called in your actions. This function is run automatically by Wheels, so most of the time, you won't need to call it explicitly in your code.

Take a look at this example action, called `display`:

```
<cffunction name="display">
    <cfset renderPage(layout="visitorLayout")>
</cffunction>
```

This assumes that you want for your action to use the layout stored at `views/blog/visitorlayout.cfm`.

The default behavior for the `layout` argument is to look for the file in the *current* controller's view folder, so here we're still assuming that the `display` action is in the `blog` controller. The `.cfm` extension will also be added automatically by Wheels, so you don't need to specifically include that part.

If you want Wheels to locate the file in a different folder, you can start the `layout` argument with a forward slash, `/`. By doing this, Wheels will know you want to start the search in the root of the `views` folder. Let's say that you're storing all miscellaneous layouts in its own folder, simply called `layouts`. You would display one of them with the following code:

```
<cffunction name="display">
  <cfset renderPage(layout="/layouts/plain")>
</cffunction>
```

Note that setting the `layout` argument on `renderPage()` will override any settings you may have made with the `usesLayout()` function. This gives you finer-grained control.

Using No Layout

If you don't want for a given template to be wrapped by a layout at all, you may want to consider creating the page as a *partial*. See the chapter about *Partials* for more information.

Another alternative is to use the `renderPage()` function and set the `layout` argument to `false`.

You can also create a separate layout that only contains the call to the `includeContent()` function in it and reference it as described above in *Using a Different Layout*. This may end up a little ugly though if you start getting a lot of small identical files like this, but the option is there for you at least.

Lastly, if your view needs to return XML code or other data for JavaScript calls, then you should reference the `renderNothing()` and `renderText()` functions to see which would be best used by your action.

Nested/Inherited Layouts

Like many templating languages, Wheels offers the ability to create layout files that can be "inherited" by other layout files. The end goal is to create a "parent" layout that has missing sections intended to be filled in by "child" layouts.

Wheels's approach involves the use of `includeLayout()` and the `contentFor()` function that we mentioned briefly earlier in this chapter.

The `includeLayout()` function simply includes another layout file. The common usage for this is to include a parent layout from a child layout.

So what about the `contentFor()` function? Well, as you may recall the code in the default layout file in Wheels contains this:

```
<cfoutput>#includeContent()#</cfoutput>
```

The `includeContent()` function accepts a `name` argument. The reason we don't have to use it above is because it defaults to `body`. This `body` variable has been set internally in the Wheels framework code with the use of the `contentFor()` function.

The point I'm trying to make here is that we can use this same functionality to set *any* type of content for, as an example, sections

in our layout files. So, armed with this new knowledge, let's create some nested layout awesomeness.

Say we have a global layout and want to fill out content in it from controller specific layouts, here's how we can do it.

The child layout:

```
<cfset contentFor(pageTitle#my custom title)>
<cfoutput#includeLayout("layout" )#cfoutput>
```

Note: If your parent file is one of the default ones named `layout.cfm` you can actually remove the "layout" string above since the default for `includeLayout()` is `layout` anyway. We're just including it here for completeness and to show that you can of course achieve this regardless of what your parent layout file is named.

The parent layout:

```
<html>
  <head>
    <title><cfoutput#includeContent("pageTitle" )#cfoutput*/title>
  </head>
  <body>
    <cfoutput#includeContent("body" )#cfoutput>
  </body>
</html>
```

Similar to above, you can remove "body" because that is the default on the `includeContent()` function.

That was a fairly basic example of how you can achieve nested layouts in Wheels to DRY up your code. You can of course expand on this by having entire sections of HTML (like a sub menu for example) be created by the child layouts. Also, as a reminder, don't forget that you can use `includePartial()` from inside your layout files as well to further keep things DRY.

Layouts for Emails and Partials

Besides having layouts for view pages in Wheels, you can also have them on emails that you send out and partial files that you include. We have chosen to speak about these in their respective chapters though: [Sending Email](#) and [Partials](#).

Form Helpers and Showing Errors

Wheels ties your application's forms together with your model layer elegantly. With Wheels form conventions, you'll find yourself spending less time writing repetitive markup to display forms and error messages.

The majority of applications are not all about back-end. There is a great deal of work to perform on the front-end as well. It can be argued that most of your users will think of the interface as the application.

Wheels is here to take you to greener pastures with its Helper Functions. Let's get visual with some code examples.

Simple Example: The Old Way

Here is a simple form for editing a user profile. Normally, you would code your web form similarly to this:

```
<cfoutput>
<form action="/profile/save"method="post ">
  <div>
    <label for=#firstName>First Name</label>
    <input id=#firstName" name="firstName" value="#profile.firstName#"/>
  </div>
  <div>
    <label for=#lastName>Last Name</label>
    <input id=#lastName" name="lastName" value="#profile.lastName#"/>
  </div>
  <div>
    <label for=#department>Department</label>
    <select id=#department" name="departmentId">
      <cfloop query=#departments">
        <option
          value=#departments.id#"
          <cfif profile.departmentId eq departments.id>
            selected="selected"
          </cfif>
          #departments.name#/option>
      </cfloop>
    </select>
  </div>
  <div>
    <input type=#hidden" name="id" value="#department.id#"/>
    <input type=#submit" value="Save Changes"/>
  </div>
</form>
</cfoutput>
```

Then you would write a script for the form that validates the data submitted, handles interactions with different data sources, and displays the form with errors that may happen as a result of user input.

We know that you are quite familiar with the drudgery of typing this sort of code over and over again. Let's not even mention the pain associated with debugging it or adding new fields and business logic!

Making Life Easier: Wheels Form Helpers

The good news is that Wheels simplifies this quite a bit for you. At first, it looks a little different using these conventions. But you'll quickly see how it all ties together and saves you some serious time.

Rewriting the Form with Wheels Conventions

Let's rewrite and then explain.

```
<cfoutput>
#startFormTag(action="save")#
  #textField(
    label="First Name", objectName="profile", property="firstName",
    prependToLabel<div>", append=*/div>", labelPlacement="before"
  )#
  #textField(
    label="Last Name", objectName="profile", property="lastName",
    prependToLabel<div>", append=*/div>", labelPlacement="before"
  )#
  #select(
    label="Department", objectName="profile", property="departmentId",
    options=departments,
    prependToLabel<div>", append=*/div>", labelPlacement="before"
  )#
  <div>
    #hiddenField(objectName="department", property="id")#
    #submitTag()#
  </div>
#endFormTag()#
</cfoutput>
```

I know what you are thinking. 9 lines of code can't replace all that work, right? In fact, they do. The HTML output will be exactly the same as the previous example. By using Wheels conventions, you are saving yourself a lot of key strokes and a great deal of time.

Factoring out Common Settings with Global Defaults

By setting up global defaults (as explained in the Configuration and Defaults for the `prependToLabel`, `append`, and `labelPlacement` arguments, you can make the form code ever simpler across your whole application.

Here are the settings that you would apply in `config/settings.cfm`:

```
<cfset
  set(
    functionName="textField", prependToLabel="<div>", append="</div>",
    labelPlacement="before"
  )
>
<cfset
  set(
    functionName="select", prependToLabel="<div>", append="</div>",
    labelPlacement="before"
  )
>
```

And here's how our example code can be simplified as a result:

```
<cfoutput>
#startFormTag(action="save")#

  #textField(label="First Name", objectName="profile", property="firstName")#
  #textField(label="Last Name", objectName="profile", property="lastName")#
  #select(
    label="Department", objectName="department", property="departmentId",
    options=departments
  )#
  <div>
    #hiddenField(
      objectName="profile", property="departmentId", options=departments
    )#
    #submitTag()#
  </div>

#endFormTag()#
</cfoutput>
```

All that the controller needs to provide at this point is a model object instance named `profile` that contains `firstName`, `lastName`, and `departmentId` properties and a query object named `departments` that contains identifier and text values. Note that the instance variable is named `profile`, though the model itself doesn't necessarily need to be named `profile`.

If you pass the form an empty instance named `profile` (for example, created by `new()`), the form will display blank values for all the fields. If you pass it an object created by a finder like `findOne()` or `findByKey()`, then the form will display the values provided through the object. This allows for us to potentially use the same view file for both create and update scenarios in our application.

Form Feedback

If you really want to secure a form, you need to do it server side. Sure, you can add JavaScript here and there to validate your web form. Unfortunately, disabling JavaScript (and thus your JavaScript-powered form validation) is simple in web browsers, and (God forbid) malicious bots tend not to listen to JavaScript.

Securing the integrity of your web forms in Wheels on the server side is very easy. Assuming that you have read the chapter on Object Validation, you can rest assured that your code is a lot more secure now.

Displaying a List of Model Validation Errors

Wheels provides you with a tool set of Helper Functions just for displaying error messages as well.

In the controller, let's say that this just happened. Your model includes validations that require the presence of both `firstName` and `lastName`. The user didn't enter either. So in the controller's `save` action, it loads the model object, sets the values that the user submitted, sees that there was a validation error after calling `save()`, and displays the form view again.

The `save` action may look something like this:

```
<cffunction name="save">
    <cfif isPost() and StructKeyExists(params,"profile")>
        <!-- In this example, we're loading a new object with the form data -->
        <cfset profile = model("UserProfile").new(params.profile)
        <cfset profile.save()

        <!-- If there were errors with the form submission,
        show the form again with errors -->
        <cfif profile.hasErrors()>
            <cfset renderPage(template="profileForm")>
        <!-- If everything validated, then send user to success message -->
        <cfelse>
            <cfset
                flashInsert(
                    success="The user profile for #profile.firstName# #profile.lastName#
                    was created successfully."
                )
            >
            <cfset redirectTo(controller=params.controller)
        </cfif>
    </cfif>
</cffunction>
```

Notice that the `profileForm` template is called if the `profile` object's `hasErrors()` method returns `true`.

Let's take the previous form example and add some visual indication to the user about what he did wrong and where, by simply adding the following code on your form page.

```
<cfoutput>
#errorMessagesFor("profile")#

#startFormTag(action="save")#

#textField(label="First Name", objectName="profile", property="firstName")#
#textField(label="Last Name", objectName="profile", property="lastName")#
#select(
    label="Department", objectName="department", property="departmentId",
    options=departments
```

```

    )#
    <div>
        #hiddenField(objectName="department", property="id")#
        #submitTag()#
    </div>
#endFormTag()#
</cfoutput>

```

How about that? With just that line of code (and the required validations on your object model), Wheels will do the following:

- Generate an HTML unordered list with a HTML class name of `errorMessages`.
- Display all the error messages on your `profile` object as list items in that unordered list.
- Wrap each of the erroneous fields in your form with a surrounding `<div class="fieldWithErrors">` HTML tag for you to enrich with your ninja CSS skills.

There is no longer the need to manually code error logic in your form markup.

Showing Individual Fields' Error Messages

Let's say that would rather display the error messages just below the failed fields (or anywhere else, for that matter). Wheels has that covered too. All that it takes is a simple line of code for each form field that could end up displaying feedback to the user.

Let's get practical and create some error messages for the `firstName` and `lastName` fields:

```

<cfoutput>
#startFormTag(action="save")#

    #textField(label="First Name", objectName="profile", property="firstName")#
    #errorMessageOn(objectName="profile", property="firstName")#

    #textField(label="Last Name", objectName="profile", property="lastName")#
    #errorMessageOn(objectName="profile", property="lastName")#

    #selectTag(
        label="Department", objectName="department", property="departmentId",
        options=departments
    )#

    <div>
        #hiddenField(
            objectName="profile", property="departmentId", options=departments
        )#
        #submitTag()#
    </div>

#endFormTag()#
</cfoutput>

```

Notice the call to the `errorMessageOn()` function below the `firstName` and `lastName` fields. That's all it takes to display the

corresponding error messages of each form control on your form.

And the error message(s) won't even display if there isn't one. That way you can yet again use the same form code for error and non-error scenarios alike.

Types of Form Helpers

There is a Wheels form helper for basically every type of form element available in HTML. And they all have the ability to be bound to Wheels model instances to make displaying values and errors easier. Here is a brief description of each helper.

Text, Password, and TextArea Fields

Text and password fields work similarly to each other. They allow you to show labels and bind to model object instances to determine whether or not a value should be pre-populated.

```
#textField(label="Username", objectName="user", property="username")#
#passwordField(label="Password", objectName="user", property="password")#
#textArea(label="Bio", objectName="user", property="biography", rows="5", cols="40")#
```

May yield the equivalent to this HTML (if we assume the global defaults defined above in the section named *Factoring out Common Settings with Global Defaults*):

```
<div>
  <label for#user-username>Username</label>
  <input id#user-username type="text" name="user[username]" value="cfguy" />
</div>
<div>
  <label for#user-password>Password</label>
  <input id#user-password type="password" name="user[password]" value=" " />
</div>
<div>
  <label for#user-biography>Bio</label>
  <textarea id#user-biography name="user[biography]" rows="5" cols="40">
    CF Guy really is a great guy. He's much nicer than .NET guy.
  </textarea>
</div>
```

(Note that we added `rows` and `cols` arguments so that we could meet XHTML compliance. They are optional arguments that only get passed through to the HTML tag if you provide them.)

Hidden Fields

Hidden fields are powered by the `hiddenField()` form helper, and it also works similarly to `textField()` and `passwordField()`.

```
#hiddenField(objectName="user", property="id")#
```

Would yield this type of markup:

```
<input type="hidden" name="user[id]" value="425" />
```

The big difference is that hidden fields do not have labels.

Select Fields

As hinted in our first example of form helpers, the `select()` function builds a `<select>` list with options. What's really cool about this helper is that it can populate the `<option>`s with values from a query, struct, or array.

Take a look at this line:

```
#select(  
    label="Department", objectName="user", property="departmentId",  
    options=departments  
)#
```

Assume that the `departments` variable passed to the `options` argument contains a query, struct, or array of department data that should be selectable in the drop-down.

Each data type has its advantages and disadvantages:

- **Queries** allow you to order your results, but you can only use one column. But this can be overcome using Calculated Properties.
- **Structs** allow you to build out static or dynamic values using whatever data that you please, but there is no guarantee that your CFML engine will honor the order in which you add the elements.
- **Arrays** also allow you to build out static or dynamic values, and there is a guarantee that your CFML engine will honor the order. But arrays are a tad more verbose to work with.

Wheels will examine the data passed to `options` and intelligently pick out elements to populate for the `<option>`s' values and text.

- **Query:** Wheels will try to pick out the first numeric column for `value` and the first non-numeric column for the display text. The order of the columns is determined how you have them defined in your database.
- **Struct:** Wheels will use the keys as the `value` and the values as the display text.
- **Array:** Wheels will react depending on how many dimensions there are. If it's only a single dimension, it will populate both the `value` and display text with the elements. When it's a 2D array, Wheels will use each item's first element as the `value` and each element's second element as the display text. For anything larger than 2 dimensions, Wheels only uses the first 2 sub-elements and ignores the rest.

Here's an example of how you might use each option:

```
<!--- Query generated in your controller --->  
<cfset departments = findAll(orderBy="name")>  
  
<!--- Hard-coded struct set up in events/onapplicationstart.cfm --->  
<cfset application.departments$1] = "Sales">  
<cfset application.departments$2] = "Marketing">
```

```

<cfset application.department$}] = "Information Technology"
<cfset application.department$#] = "Human Resources"

<!-- Array built from query call in model -->
<cfset departments = this.findAll(orderBy=lastName,hq)>
<cfset departmentsArray = []
<cfloop query=#departments#
    <cfset newDept = [department.id, departments.name' & " & departments.hq]
    <cfset ArrayAppend(departmentsArray, newDept)
</cfloop>
<cfreturn departments>

```

When sending a query, if you need to populate your <option>s' values and display text with specific columns, you should pass the names of the columns to use as the `textField` and `valueField` arguments.

You can also include a blank option by passing `true` or the desired text to the `includeBlank` argument.

Here's a full usage with this new knowledge:

```

#select(
    label="Department", objectName="user", property="departmentId",
    options=departments, valueField="id", textField="departmentName",
    includeBlank="Select a Department"
)#

```

Radio Buttons

Radio buttons via `radioButton()` also take `objectName` and `property` values, and they accept an argument called `tagValue` that determines what value should be passed based on what the user selects.

Here is an example using a query object called `eyeColor` to power the possible values:

```

<fieldset>
  <legend>Eye Color</legend>
  <cfloop query=#eyeColor#
    #radioButton(
      label=eyeColor.color, objectName="profile", property="eyeColorId",
      tagValue=eyeColor.id, labelPlacement="after"
    )#br />
  </cfloop>
</fieldset>

```

If the `profile` object already has a value set for `eyeColorId`, then `radioButton()` will make sure that that value is checked on page load.

If `profile.eyeColorId`'s value were already set to 1, the rendered HTML would appear similar to this:

```

<fieldset>
  <legend>Eye Color</legend>
  <input type="radio" id="profile-eyeColorId-2" name="profile[eyeColorId]" value="2" />
  <label for=#profile-eyeColorId-2#Blue</label>#br />
  <input
    type="radio" id="profile-eyeColorId-1" name="profile[eyeColorId]" value="1"

```

```

        checked="checked"
    />
    <label for=#profile-eyeColorId-1>Brown</label><br />
    <input type="radio" id="profile-eyeColorId-3" name="profile[eyeColorId]" value="3" />
    <label for=#profile-eyeColorId-3>Hazel</label><br />
</fieldset>

```

Note that if you don't specify `labelPlacement="after"` in your calls to `radioButton()`, Wheels will place the labels before the form controls.

Check Boxes

Check boxes work similarly to radio buttons, except `checkBox()` takes parameters called `checkedValue` and `uncheckedValue` to determine whether or not the check box should be checked on load.

Note that binding check boxes to model objects is best suited for properties in your object that have a yes/no or true/false type value.

```

#checkBox(
  label="Sign me up for the email newsletter.",
  objectName="customer", property="newsletterSubscription", labelPlacement="after"
)#

```

Because the concept of check boxes don't tie to well to models (you can select several for the same "property"), we recommend using `checkBoxTag()` instead if you want to use check boxes for more values than just true/false. See the *Helpers That Aren't Bound to Model Objects* section below.

File Fields

The `fileField()` helper builds a file field form control based on the supplied `objectName` and `property`.

```

#fileField(label="Photo", objectName="profile", property="photo")#

```

In order for your form to pass the correct `enctype`, you can pass `multipart=true` to `startFormTag()`:

```

#startFormTag(action="save", multipart=true)#

```

Helpers That Aren't Bound to Model Objects

Sometimes you'll want to output a form element that isn't bound to a model object.

A search form that passes the user's query as a variable in the URL called `q` is a good example. In this example case, you would use the `textFieldTag()` function to produce the `<input>` tag needed.

```

#textFieldTag(label="Search", name="q", value=params.q)#

```

There are "tag" versions of all of the form helpers that we've listed in this chapter. As a rule of thumb, add `Tag` to the end of the function name and use the `name` and `value`, `checked`, and `selected` arguments instead of the `objectName` and `property` arguments that you normally use.

Passing Extra Arguments for HTML Attributes

Much like Wheels's `linkTo()` function, any extra arguments that you pass to form helpers will be passed to the corresponding HTML tag as attributes.

For example, if we wanted to define a `class` on our starting form tag, we just pass that as an extra argument to `startFormTag()`:

```
■ #startFormTag(action="save", class="login-form")#
```

Which would produce this HTML:

```
■ <form action="/user/save" class="login-form">
```

Special Form Helpers

Wheels provides a few extra form helpers that make it easier for you to generate accessible fields for dates and/or times. These also bind to properties that are of type `DATE`, `TIMESTAMP`, `DATETIME`, etc.

We won't go over these in detail, but here is a list of the date and time form helpers available:

- `dateSelect()`
- `dateSelectTags()`
- `timeSelect()`
- `timeSelectTags()`
- `dateTimeSelect()`
- `dateTimeSelectTags()`
- `yearSelectTag()`
- `monthSelectTag()`
- `daySelectTag()`
- `hourSelectTag()`
- `minuteSelectTag()`
- `secondSelectTag()`

Displaying Links for Pagination

How to create links to other pages in your paginated data in your views.

In the chapter titled Getting Paginated Data, we talked about how to get pages of records from the database (records 11-20, for example). Now we'll show you how to create links to the other pages in your view.

Displaying Paginated Links with the `paginationLinks` Function

If you have fetched a paginated query in your controller (normally done using `findAll()` and the `page` argument), all you have to do to get the page links to show up is this:

```
<cfoutput#paginationLinks()#>
```

Given that you have only fetched one paginated query in your controller, this will output the links for that query using some sensible defaults.

How simple is that?

Arguments Used for Customization

Simple is good, but sometimes you want a little more control over how the links are displayed. You can control the output of the links in a number of different ways. We'll show you the most important ones here. Please refer to the `paginationLinks()` documentation for all other uses.

The `name` Argument

By default, Wheels will create all links with `page` as the variable that holds the page numbers. So the HTML code will look something like this:

```
<a href="/main/userlisting?page=1">
<a href="/main/userlisting?page=2">
<a href="/main/userlisting?page=3">
```

To change `page` to something else, you use the `name` argument like so:

```
<cfoutput#paginationLinks(name="pgnum")#>
```

By the way, perhaps you noticed how Wheels chose to use that hideous question mark in the URL, despite the fact that you have URL rewriting turned on? Because `paginationLinks()` uses `linkTo()` in the background, you can easily get rid of it by creating a custom route. You can read more about this in the Using Routes chapter.

The `windowSize` Argument

This controls how many links to show around the current page. If you are currently displaying page 6 and pass in `windowSize=3`, Wheels will generate links to pages 3, 4, 5, 6, 7, 8, and 9 (three on each side of the current page).

The `alwaysShowAnchors` Argument

If you pass in `true` here, it means that no matter where you currently are in the pagination or how many page numbers exist in total, links to the first and last page will always be visible.

Managing More Than One Paginated Query Per Page

Most of the time, you'll only deal with one paginated query per page. But in those cases where you need to get/show more than one paginated query, you can use the `handle` argument to tell Wheels which query it is that you are referring to.

This argument has to be passed in to both the `findAll()` function and the `paginationLinks()` function. (You assign a handle name in the `findAll()` function and then request the data for it in `paginationLinks()`.)

Here is an example of using handles:

In the controller...

```
<cfset users = model("user").findAll(handle="userQuery", page=params.page, perPage=25)
<cfset blogs = model("blog").findAll(handle="blogQuery", page=params.page, perPage=25)
```

In the view...

```
<ul>
  <cfoutput query="users">
    <li>#users.name#/li>
  </cfoutput>
</ul>
<cfoutput #paginationLinks(handle="userQuery")#foutput>

<cfoutput query="blog">
  #address#br />
</cfoutput>
<cfoutput #paginationLinks(handle="blogQuery")#foutput>
```

That's all you need to know about showing pagination links to get you started. As always, the best way to learn how the view functions work is to just play around with the arguments and see what HTML is produced.

Date, Media, and Text Helpers

Wheels includes a plethora of view helpers to help you transform data into a format more easily consumed by your applications' users.

Wheels's included view helper functions can help you out in those tricky little tasks that need to be performed in the front-end of your web applications. Although they are called miscellaneous, they are in fact categorized into 3 categories:

- Date Helpers
- Media Helpers
- Text Helpers

We also have separate chapters about Wheels form helpers in Form Helpers and Showing Errors and creating your own helpers in Creating Your Own View Helpers.

Date Helpers

Wheels does a good job at simplifying the not so fun task of date and time transformations.

Let's say that you have a comment section in your application, which shows the title, comment, and date/time of its publication. In the old days, your code would have looked something like this:

```
<cfoutput query=#comments>
  <div class=#comment>
    <h2>#comments.title#/h2>
    <p class=#timestamp>
      #DateFormat(comments.createdAt, "mmmm d, yyyy")#
      #LCase(TimeFormat(comments.createdAt, "h:mm t"))#
    <p>#comments.comment#/p>
  </div>
</cfoutput>
```

That works, but it's pretty tedious. And if you think about it, the date will be formatted in a way that is not that meaningful to the end user.

Instead of "April 27, 2009 10:10 pm," it may be more helpful to display "a few minutes ago" or "2 hours ago." This can be accomplished with a Wheels date helper called `timeAgoInWords()`.

```
<cfoutput query=#comments>
  <div class=#comment>
    <h2>#comments.title#/h2>
    <p class=#timestamp>(#timeAgoInWords(comments.createdAt))#>
    <p>#comments.comment#/p>
  </div>
</cfoutput>
```

With that minimal change, you have a prettier presentation for your end users. And most important of all, it didn't require you to do anything fancy in your code.

Media Helpers

Working with media is also a walk in the park with Wheels. Let's jump into a few quick examples.

Style Sheets

First, to include CSS files in your layout, you can use the `styleSheetLinkTag()` function:

```
<!-- layout.cfm --->
<cfoutput>
  #styleSheetLinkTag("main")#
</cfoutput>
```

This will generate the `<link>` tag for you with everything needed to include the file at `stylesheets/main.css`.

If you need to include more than one style sheet and change the media type to "print" for another, there are arguments for that as well:

```
#styleSheetLinkTag(sources="main,blog")#
#styleSheetLinkTag(source="printer", media="print")#
```

Lastly, you can also link to stylesheets at a different domain or subdomain by specifying the full URL:

```
#styleSheetLinkTag(
  source="http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.0/themes/cupertino/jquery-ui.css"
)#
```

JavaScript Files

Including JavaScript files is just as simple with the `javascriptIncludeTag()` helper. This time, files are referenced from the `javascripts` folder.

```
#javascriptIncludeTag("jquery")#
```

Like with style sheets, you can also specify lists of JavaScript includes as well as full URLs:

```
#javascriptIncludeTag("https://ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min.js")#
```

Displaying Images

Wheels's `imageTag()` helper also provides some simple, yet powerful functionality:

```
<cfoutput>
    #imageTag("logo.png")#
</cfoutput>
```

With this simple call, Wheels will generate the `` tag for `images/logo.png` and also set the `width`, `height` and `alt` attributes automatically for you (based on image dimensions and the image file name). Wheels will also cache this information for later use in your application.

If you need to override the `alt` attribute for better accessibility, you can still do that too:

```
#imageTag(source="logo.png", alt="ColdFusion on Wheels")#
```

Text Helpers

To illustrate what the text helpers can help you with, let's see a piece of code that includes 2 of the text helpers in a simple search results page.

```
<!-- Query of search results -->
<cfparam name=searchResults type=query>
<!-- Search query provided by user -->
<cfparam name=params.q type=string>

<cfoutput>
    <p>
        #highlight(text="Your search for #params.q#", phrases=params.q)#
        returned #searchResults.RecordCount#
        #pluralize(word="result", count=searchResults.RecordCount)#.
    </p>
</cfoutput>
```

That code will highlight all occurrences of `params.q` and will pluralize the word "result" to "results" if the number of records in `searchResults` is greater than 1. How about them apples? No `<cfif>` statements, no extra lines, no nothing.

The functions we have shown in this chapter are only the tip of the iceberg when it comes to helper functions. There's plenty more, so don't forget to check out the View Helper Functions API.

Creating Your Own View Helpers

Clean up your views by moving common functionality into helper functions.

As you probably know already, Wheels gives you a lot of helper functions that you can use in your view pages.

Perhaps what you didn't know was that you can also create your own view helper functions and have Wheels automatically make them available to you. To do this, you store your UDFs (User Defined Functions) in different controller-level helper files.

The `views/helpers.cfm` File

Once a UDF is placed in this file, it will be available for use in all your views.

Alternatively, if you only need a set of functions in a specific controller of your application, you can make them controller-specific. This is done by placing a `helpers.cfm` file inside the controller's view folder.

So if we wanted a set of helpers to generally only be available for your `users` controller, you would store the UDFs in this file:

```
■ views/users/helpers.cfm
```

Any functions in that file will now only be included for the view pages of that specific controller.

When *not* to Use Helper Functions

Helper functions, together with the use of `Partials`, gives you a way to keep your code nice and DRY, but there are a few things to keep in mind as you work with them.

The `helpers.cfm` files are only meant to be used for views, hence the placement in the `views` folder.

If you need to share non-view functionality across controllers, then those should be placed in the parent controller file, i.e. `controllers/Controller.cfc`. If you need helper type functionality within a **single** controller file, you can just add it as a function in that controller and make it private so that it can't be called as an action (and as a reminder to yourself of its general purpose as well).

The same applies to reusable model functionality: use the parent file, `models/Model.cfc`. Private functions within your children models work well here, just like with controllers.

If you need to share a function globally across your entire application, regardless of which MVC layer that will be accessing it, then you can place it in the `events/functions.cfm` file.

The Difference Between `Partials` and `Helpers`

Both `partials` and `helpers` are there to assist you in keeping programmatic details out of your views as much as possible. Both do

the job well, and which one you choose is just a matter of preference.

Generally speaking, it probably makes most sense to use partials when you're generating a lot of HTML and helpers when you're not.

