CHAPTER 1

The Search for the Holy Grail(s)

Grails is one of the upcoming new wave of dynamic language-based web frameworks. In this chapter we'll explore the history behind Grails' inception and why you should be interested in it and its alternatives.

Grails specifically is a next-generation web application framework for Java that leverages the Groovy language to promote simplicity and convention. Grails leverages as many existing projects as possible to create a full stack framework that is in stark contrast to many existing frameworks for Java that present the user with an anemic[1] API.

Grails arrives at a time when web applications are getting ever more complex with the onset of Web 2.0 and Ajax, a topic I'll be discussing in greater detail later in the book. Before we get ahead of ourselves, let's explore the reasoning and user experiences of other frameworks that have led to the emergence of frameworks such as Grails.

Trouble in Paradise

Have you ever wanted a Java web application framework that required less configuration? Maybe one that magically reads your mind and guesses what you want to achieve based on the business logic rather than mountains of surrounding configuration. The amount of work it takes to develop web applications in current Java web application frameworks has seen a dramatic rise in code-generation utilities whose sole purpose is to generate surrounding configuration or jump-start application development. If you're reading this book as a developer interested in Grails I hazard a guess that you've experienced the often cumbersome multistep process it can be to work with web frameworks on the Java platform.

Trust me though, you're not alone. The search has been on for years to transform the landscape of the way we develop web applications with the Java platform and extends way beyond just the frameworks themselves, but into the entire stack. It started with the initial specification of the much heralded Java 2 Enterprise Edition (J2EE) stack, described lovingly by Bruce Tate[2] as an "elephant." The specification created a configuration-heavy, time-consuming environment, and acknowledgement of these mistakes were made particular in the area of Enterprise JavaBeans (EJB).

1. Anemic APIs are a common antipattern encapsulated by Martin Fowler in an article about domain models titled "Anemic Domain Model" (`http://www.martinfowler.com/bliki/AnemicDomainModel.html`).
2. Bruce Tate is a big critic of Java's current development process and has written a number of books on the topic, including *Bitter Java*, *Better, Faster, Lighter Java*, and *Beyond Java*. His article on the "elephant" can be found at `http://today.java.net/pub/a/today/2004/06/15/ejb3.html`.

Before the mistakes could be corrected, the open source community reacted with frameworks such as Spring, inspired by Rod Johnson's book *J2EE Development Without EJB,* allowing simplified development models, while ease of use in the object-relational mapping (ORM) arena was driven by simplified POJO (Plain Old Java Object) frameworks such as Hibernate.

It was not, however, the statically typed Java world that prompted the real innovation in web framework design. Dynamic languages were already gaining prominence in other arenas, but thanks to frameworks such as Ruby on Rails (henceforth simply referred to as Rails), Django, and TurboGears, particular focus has been placed on their usage as web application frameworks. After years of being ridiculed as toys, dynamic languages finally started to turn heads and kick-start a rethink into how web applications can be simplified.

Built with scripting languages such as Ruby and Python, this new generation of web frameworks introduced the concept of "convention over configuration" in an object-oriented environment. Instead of configuring everything via reams of XML, the convention within the files or code itself dictated how the application was configured. Suddenly there was no need to configure, and with that, the time spent on a typical development cycle was dramatically reduced. Given their dynamic nature, the frameworks in question provided the ability to save a particular class or resource and have the changes automatically propagated without any need to restart an application server. Applications could be developed with a save/reload paradigm, and state could be maintained; agile and iterative development had finally arrived.

Since much of the functionality of these frameworks can be configured and dispatched at runtime, the available APIs are significantly richer and often more domain-specific.

Scripting languages themselves have always been popular for the development of web applications because of their ability to realize changes immediately (no deployment step) and their often more simple and concise syntax. Unfortunately they've rarely been seen as appropriate for usage on large-scale systems, usually due to a lack of object-oriented features and issues with performance and scalability and/or legacy integration.

Given the advancements in the platforms in which these languages operate and the hardware on which they are executing, these issues have become less and less of a problem. Modern, dynamic virtual machines (VMs) suffer from fewer of these deficiencies and have become a real viable alternative, attracting many Java developers away from the Java platform.

The Arrival of Web 2.0 Applications

Over the past few years the web has seen significant change. The Internet bubble burst, search became the doorway to the web, and the rise of weblogs and open information saw the web become even more opinionated. On the technology front, one of the most significant changes to occur was the appearance of web applications that come close to rivaling the desktop environment. Web developers had long been dismissed as mere hackers compared to their counterparts in the desktop GUI programming world. But this was all about to change, and it all started with essentially one object: the `XMLHttpRequest`.

The `XMLHttpRequest` object has, in reality, been around for quite some time. Microsoft started the revolution by including it as a custom ActiveX object to support its Outlook web access service that integrated with Microsoft's Exchange Server.

Essentially it allowed the client (the browser) to query the server without the necessity to reload the whole page. It achieved this feat by sending an *asynchronous* request to the server via JavaScript. Although, strictly speaking, anything could be sent and received using the technology, XML was the initial technique used. Web applications could suddenly be event-driven,

use drag-and-drop, perform autocompletion that integrates with server components, and do all sorts of fancy things previously only seen in frameworks such as Swing. Applications such as Gmail and Flickr even prompted users to stop using their desktop equivalents for mail and photo organization, respectively.

Since then, the word *Ajax* (Asynchronous JavaScript and XML) was coined and the web has been abuzz with it ever since; Web 2.0 was born. The significance of this technology cannot be underestimated. It has brought capabilities to web browsers thought only possible in desktop environments. However, it has also increased the complexity, development time, and expertise required to develop web applications.

Luckily, with this influx of software engineering expertise into the client-side programming world, JavaScript libraries have advanced at a staggering pace, adopting object-oriented techniques to create simple reusable components. However, it is not just the client side that has had to adapt; a much unwanted side effect of Ajax applications is the increased number of requests that they produce. Ajax applications are for the most part loathed by network administrators because of the load placed on the servers, but equally undesirable for developers because the frameworks and tools we know and love are designed around a sequential request/response model.

Furthermore, existing Java frameworks have followed the traditional *develop-compile-deploy* paradigm, a process that becomes slower as the number of requests increase and the size of the project grows. Since Ajax applications inevitably end up being larger, in terms of project size, than traditional web applications, this has huge implications on the compilation and packaging time required to deploy a web application.

To compound these already significant factors, testing rich application flows started to become problematic. With much of the state held on the client, having to reload your browser due to an application restart could set you several steps back in any given use case. The reality was that the frameworks needed to adapt. Adapting is not the easiest thing to achieve in the statically typed world of the Java Virtual Machine (JVM) where classes tend to be compiled once. This is in complete contrast to dynamic languages that follow either an interpreted model or automatically update resources for you at run time.

The timing of the arrival of Ajax was perfect for the aforementioned dynamic-language-based frameworks as they began offering solutions that wowed developers with support for developing Ajax applications and automatic reloading. Tight integration with JavaScript libraries such as Prototype and Script.aculo.us allowed developers to easily perform asynchronous requests, implement element observers, and perform interesting effects.

In addition, features such as the iterative nature of development, no configuration or deployment cycle, and the convention-based approach meant that it was an extremely appealing environment to develop with.

Are the changing times a sign of not only a shift in the way we build applications, but also the technologies we use to build them? Is this the end of Java as we know it? And is Java venturing down the road to become the next COBOL? Many believe so; but the reality is quite different.

The Power of Java

Unfortunately, however wonderful these dynamic frameworks are, they remain inaccessible to thousands upon thousands of organizations for one simple reason: they do not integrate seamlessly with Java. Java integration itself goes far beyond simply executing within the context of the JVM. It means API integration, architectural integration (security, profiling, debugging, etc.),

object model integration, Java Enterprise Edition integration, and knowledge and mindshare integration (yes company X already has dozens of Java developers if not more).

Organizations across the world have a significant investment in Java technology as a platform for their business. They've invested in application servers, support contracts, training materials, employees, and hardware optimized for the JVM. All of this amounts to too much to just walk away from to embrace the latest hot framework.

The Java industry is a huge multibillion dollar behemoth that encompasses not only web applications but server applications, desktop applications, smart cards, handheld devices, mobile phones, set-top boxes—this list could go on forever. Conferences dedicated to Java are some of the largest technology events in the world, proving to be the envy of other technology industries.

The size of the industry is sometimes difficult to fathom and it's built on the strength of the *platform*—a platform that has ensured the Java language has more open source libraries than any other language on the planet.

Need a library to programmatically create PDF documents? You have FOP or iText. Want a choice of web frameworks that cover every potential use case? Java has dozens. Need a platform to build service-oriented architecture (SOA) applications? The Java platform has the solution for you.

For many, Java and its sibling in the enterprise world, Java Enterprise Edition, have been the platform of choice, having reached an unrivalled level of maturity and industry support. Java application servers have become robust, scalable managed environments with advanced deployment capabilities, security controls, and web services integration.

Unfortunately, with all this power, Java web applications can be a pain to develop when compared to their dynamic rivals from Ruby, Python, and PHP. Much of the time involved in creating one of these beasts is spent on configuration, the build process, and deployment. Java developers have strived to create build automation tools that make this process easier, including tools that generate source code. This has improved even further with the advent of Java 5 annotations support. But developing Java web applications for the most part remains quite a complex, configuration-heavy experience.

Nevertheless it would be simply unjust if the Java community were denied access to a framework with the same capability as those available on dynamic language platforms.

Since there are several dynamic languages that run happily on top of the JVM, such as Jython, JRuby, and Groovy, it seems imminently possible. Luckily the Java community tends to take the good ideas and make them better; enter Grails.

Grails: The Story So Far

The formation of Grails came on the back of the fabulous progress made in the Groovy language in the summer of 2005. A small group of Groovy enthusiasts who had been astounded by the power of dynamic languages, but unable to take advantage of them due to an existing investment in Java, formed the then-named Groovy on Rails in honor of Ruby on Rails. The Groovy language itself was finally reaching a level of maturity where it was usable as a dynamic language in enterprise environments, making the timing of this formation perfect.

The goal of Grails was to go beyond what other languages and their associated frameworks could offer in the web application space. Grails aimed to do the following:

- Integrate tightly with the Java platform
- Be simple on the surface but retain flexibility to access the powerful underlying Java frameworks and features
- Learn from the mistakes already made on the mature Java platform

By utilizing Groovy as the starting point for the framework, it gave Grails a huge head start. Groovy's goal was to create a language that is a seamless transition for Java developers into the dynamically typed scripting world, bringing advanced features impossible to implement with statically typed languages.

Groovy's goals as a general-purpose language for the Java platform are very much inline with those of Grails as a web framework: to make the transition into the dynamically typed world as painless a learning experience as possible.

The creators of Groovy recognized that for Java developers to be truly productive when using a dynamic language it should not require a huge mental shift to go from language to language. Thanks to this, Groovy uses a strikingly similar syntax and the same APIs available to you in the JDK.

Groovy compiles directly down to byte code, thus ensuring that it also shares the same object model as that used within the JVM. A Groovy object *is* a Java object and does not have any specialized interpreter or VM.

Java Integration

Groovy's ability to seamlessly integrate with Java, along with its Java-like syntax, is the No. 1 reason so much hype was generated around its conception. Here we had a language with similar capabilities to languages such as Ruby and Smalltalk running directly in the JVM. The potential is obvious, and the ability to intermingle Java code with dynamic Groovy code is huge.

In addition, Groovy allowed you to mix static types and dynamic types, providing the safety of static types with the power and flexibility to opt out of using static typing where deemed necessary.

This level of Java integration is what drives Groovy's continued popularity, and the world of web applications is no different. Across different programming platforms there are varying idioms to express essentially the same concept. In the Java world we have servlets, filters, tag libraries, and Java Server Pages (JSP). Moving to a new platform requires relearning all of these concepts and their equivalent APIs or idioms—easy for some, a challenge for others.

Not that learning new things is bad, but the problem is that there is a cost attached to knowledge gain in the real world, and this can be a major stumbling block in the adoption of any new technology that deviates from the standards or conventions defined within the Java platform and the enterprise.

In addition, there are standards within Java for deployment, management, security, naming, and more. The goal of Grails is to create a platform with the essence of frameworks like Ruby on Rails that embraces the mature environment that is the Java Enterprise Edition and associated APIs.

Simplicity and Power

Clearly embracing all of these wonderful features within the Java platform should not come at the cost of simplicity, and this is where the expressiveness of Groovy really shines through.

Groovy is one of the very few languages available on the Java platform that provides both tight Java integration and syntactic expressiveness. However, frequently, simplicity and convention will only get you so far. Grails aims to provide the flexibility to leverage the underlying "power features" of the Java platform.

To ensure this flexibility is available, careful choices were made regarding technologies that would power Grails. Reinvention of the wheel is not a phrase that sits well in the Java community, and hence the underlying infrastructure within Grails is powered by the most popular open source technologies in their respective categories:

Hibernate: The de facto standard for ORM in the Java world

Spring: The hugely popular open source Inversion of Control (IoC) container and wrapper framework for Java

Quartz: An enterprise-ready, job-scheduling framework allowing flexibility and durable execution of scheduled tasks

SiteMesh: A robust and stable layout-rendering framework

For some readers the concept of ORM and IoC may seem a little alien. As an explanation, ORM simply serves as a way to map objects from the object-oriented world onto tables in a relational database. ORM provides an additional abstraction above SQL, allowing developers to think about their domain model instead of getting wrapped up in reams of SQL.

IoC, also known as *dependency injection*, is a way of "wiring" together objects so that their dependencies are available at run time. As an example, an object that performs persistence may require access to a data source. IoC provides a way to take the responsibility of obtaining a reference to the data source off the developer. Nevertheless, don't get too wrapped up in these concepts for the moment, as their usage will become clear later in the book.

Moving on, Grails exposes each of the aforementioned frameworks capabilities via a simplified interface, but still allows the usage of them using their documented configuration and development capabilities. Figure 1-1 illustrates how Grails relates to these frameworks and the Java enterprise stack.

At Grails' core lies the JVM, which both the Java and Groovy languages compile to via byte code. The leveraged frameworks, such as Spring, Hibernate, and Quartz (to name a few), are built on the strength of the Java language and the JVM. Groovy can work with these APIs outside of Grails, but Grails harnesses Groovy's advanced features combined with the Java Enterprise Edition environment to provide a simplified environment for building web applications.

The APIs for many of the frameworks depicted in Figure 1-1 are often criticized for being overly complex even though they're often spoken of as representing a *lightweight* approach to Java web application development. One of the primary aims of Grails is to provide an additional abstraction layer over these frameworks that takes advantage of the dynamic nature of Groovy.

However, the full underlying power of these frameworks is still readily available to harness should you so choose. Having all this power is one thing, but it would be rather silly if Grails didn't learn from some of the mistakes in previous generations of web frameworks.

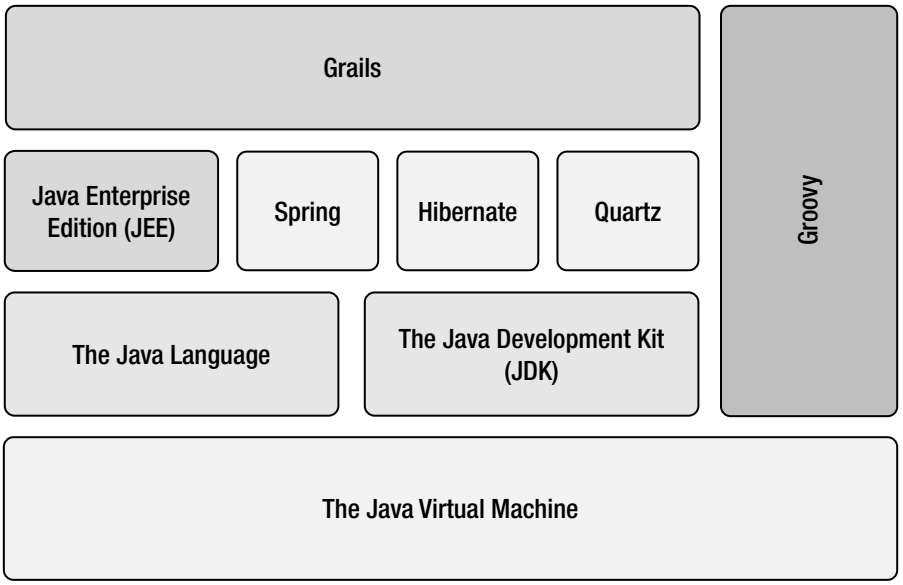

Figure 1-1. *How Grails stacks up*

Lessons Learned

Java web development practices and methodologies have been refined and optimized over a number of years. Mistakes have been made in both the frameworks and the specifications themselves. These have been slowly rectified over time in their various revisions based on the experiences and feedback from the Java community.

The dynamic language platforms are not entirely devoid of these mistakes themselves—things like cleanly separating logic so that the view does not contain scriptlet code, and providing a clean, clear model-view-controller (MVC) architecture where business logic is separated from view logic. The MVC architecture is designed to minimize this risk, but clearly it is still possible to fall into this trap within the constraints of an MVC application.

Grails aims to provide the necessary infrastructure to cleanly separate this logic and includes concepts more familiar in the Java space such as tag libraries, a service tier, and domain-driven development.

Why You Should Be Interested in Grails

So you've read a bit of the background, but the question is, why should you be interested in using it? Every solution is born out of a problem. The repetitive nature of web development and the common issues within it, such as application state, converting from text to object representation, and dealing with its multithreaded nature, are just some of the things faced by today's web application developers.

These are only growing more complex as the usage of technologies such as Ajax increase the complexity of the application state and push the number of requests into the stratosphere. A dynamic framework can help in reducing the strain of the development cycle at a more simplistic level.

However, there is a lot to be said for the benefits of static typing, advanced IDE support, and refactoring that is available on the Java platform and its associated development environments. As your application grows in complexity you begin to realize just how important these features are. Grails allows a blended approach by mixing both statically typed Java code and the dynamic nature of a language such as Groovy.

To this end, Grails allows you to scale up your application as it grows—scaling not in terms of performance, but in terms of *application complexity*. Have a particular piece of logic that is better suited to a Java implementation? No problem. Groovy and Grails work seamlessly with Java to enable this and thus allow you to continue using IDEs such as Eclipse and IntelliJ IDEA for code navigation, analysis, and refactoring.

Grails could be the solution that you've been looking for in the Java space. Even if you're already developing in one of the other aforementioned frameworks, Grails is worth a look because of the power and flexibility it offers, the ability to use a blended approach by mixing static and dynamic typing, and tight integration with the Java platform.

In addition, being able to integrate with Java and the JVM is only part of the story. Why dump all your knowledge of frameworks such as Spring and Hibernate? Grails is built on top of these, and their existing APIs are fully available for you to call just as in Java code. These frameworks are also written in Java themselves, so you get the benefit of that in terms of the performance they offer over their rivals.

Heard enough? Can't wait to get started? Let's start our journey into the Grails universe by installing it first.

Getting Started with Grails

As with any software, the first thing you need to do with Grails is install it. To do so follow these steps:

1. Download the latest Grails distribution from the web site `grails.org`.
2. Extract the relevant archive into a convenient location. If you're organized, this could be in a development sandbox or even simply on your desktop.
3. Set the `GRAILS_HOME` environment variable to the location where you extracted the Grails distribution. Setting environment variables is a rather platform-specific activity, but in Windows this can be via the Advanced tab in the properties of My Computer. Alternatively, if you're running Mac OS X you could add a `GRAILS_HOME` variable to your `.profile` file within your home directory (i.e., `cd ~`).

4. In addition, to get the Grails scripts to work from a terminal or command window you need to add the `bin` directory within Grails to your `PATH` that will make the Grails scripts visible to the window. To do this, take these steps:

   a. On Windows append `%GRAILS_HOME%\bin` to the `Path` variable.

   b. On Unix append `$GRAILS_HOME/bin` to the `PATH` variable.

To validate your installation, open a command window and type the command **grails**. If you have successfully installed Grails, the command will output the usage help executed in Listing 1-1.

■ **Note** This listing introduces a convention that will be used throughout the book by using **bold** to highlight user input.

Listing 1-1. *Grails Command-Line Help*

```
>grails
```

```
help:
 Usage: grails [target]

Targets:
"create-app"          - Create a new grails app
"create-controller"   - Create a new controller
"create-service"      - Create a new service
"create-domain-class" - Create a new domain class
"create-taglib"       - Create a new tag library class
"create-test-suite"   - Create a new test suite
"create-job"          - Create a quartz scheduled job
"generate-controller" - Generates a controller from a domain class
"generate-views"      - Generates the views from a domain class
"generate-all"        - Generates all artifacts from a domain class
"test-app"            - Run current app's unit tests
"run-app"             - Run the application locally and wait
"create-webtest"      - Create the functional test layout
"run-webtest"         - Run the functional tests for a running app
"shell"               - Opens the Grails interactive command line shell
"console"             - Opens the Grails interactive swing console
"war"                 - Create a deployable Web Application Archive (WAR)
```

It may be interesting to note at this point that Grails uses the Apache Ant (`http://ant.apache.org`) build system to power these targets, and throughout the book I will be referring to them as *targets* (as opposed to *commands* or *operations*).

■ **Note** Ant is supported by every modern IDE on the market and is the ubiquitous build system for the Java platform. The advantage being that there is an extremely high level of knowledge of Ant within the Java community, making the Grails build easily customizable if additional functionality is required by your development process.

Now that you've seen what targets are available, let's find out how to run them. Running the targets is a prerequisite to be able to effectively use Grails, so getting this step right will benefit you greatly in the long run. Luckily, it's pretty simple, as you'll see next.

Running the Targets

To execute the Grails targets you simply have to run the `grails` command followed by the name of the target. For example, to create a new Grails application, you could run the `grails create-app` command.

An additional feature of Ant is that you can combine targets. For example, if you need to test your application and then create a web application archive (WAR) file for deployment onto an application server you could type this:

```
grails test-app war
```

What this will do is execute the `test-app` target first and if successful will continue on to the `war` target. This becomes more useful as you get familiar with all the targets available and how they can be combined. It may be worth conducting some experiments of your own to get an idea of what's possible.

Looking back at the list of targets, the `create-*` targets are convenience targets for setting up a Grails application and creating Grails artifacts. However, unlike a few other frameworks, the targets themselves don't perform any special configuration of their own in the background. This is significant as it allows you to start off using the `create-*` targets as a learning tool to advance your knowledge of Grails.

Then once you are familiar enough with where everything goes you can create the various classes yourself using your favorite IDE if you so choose. Moving on, the `generate-*` targets are extremely powerful, allowing the generation of boilerplate code, which is great for both the learning process and getting started quickly. These will be discussed in detail in Chapter 5, which delves into a concept called *scaffolding*.

Of the remaining targets, the most useful are those for executing the Grails application (using the included Jetty application server) for running unit and functional web tests and creating web application archives.

The Obligatory "Hello World!"

So let's get started creating our first Grails application. As is traditional, no book would be complete without a "Hello World!" example, so here it is. To complete your journey through creating a "Hello World!" application you're going to step through the following tasks:

1. Execute the `grails create-app` command to create a basic Grails application.
2. Create something called a *controller* that will handle a web request.
3. Use the controller to display some text.
4. Run the Grails application and view what you achieved in a web browser.

So let's get going. First up is the creation of a project. Grails provides a target to do just this. Having a common project infrastructure across projects is hugely beneficial, as it ensures newcomers to a project who are already familiar with the way Grails projects are laid out can get up to speed quickly. This allows a developer to immediately focus his attention on the code instead of focusing on understanding how a project fits together.

Projects such as Maven (`http://maven.apache.org`) have introduced this concept in the Java world. Unfortunately the vast majority of projects still use a custom build system via Ant. It represents one of the graver mistakes made by the designers of the Java Enterprise Edition specification because every Java web project tends to have a different structure and resource layout. Needless to say, this is one mistake that can't be corrected easily, hence Grails ensures that projects are structured in a common way.

To create our "hello" application we need to run the `create-app` target. This will prompt you for the name of the application you wish to create. Enter the word **hello** and hit the return key as per the example in Listing 1-2.

Listing 1-2. *Running the create-app Target*

```
>grails create-app
init-props:

create-app:
    [input] Enter application name:
hello
```

Upon completion, the target will have created the "hello" Grails application and the necessary directory structure. The next step is to navigate to the newly created application in the command window using the shell command:

```
cd hello
```

At this point you have a clean slate—a newly created Grails application—with the default settings in place. A screenshot of the structure of a Grails application can be seen in Figure 1-2.

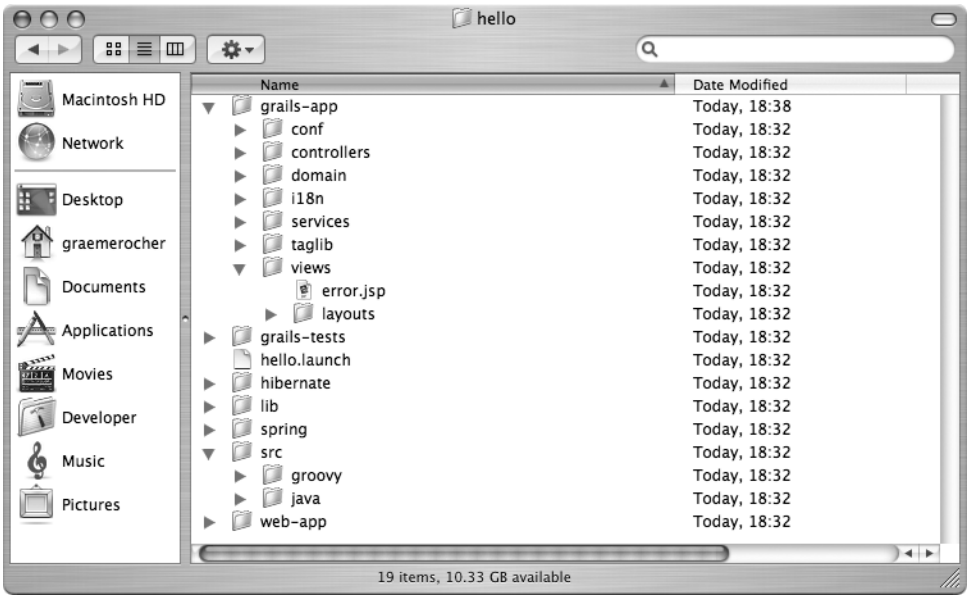

Figure 1-2. *The "hello" application structure*

We will delve more into the details of the structure of a Grails application and the roles of the various files and directories in Chapter 3. You will notice, however, how Grails contains directories for controllers, domain objects (models), and views.

Grails is an MVC framework and has models, views, and controllers to cleanly separate concerns. Nevertheless, to display a simple "Hello World!" we only need to be concerned with controllers for the moment.

Controllers are core to any MVC framework, Grails being no different; and in fact the only way to handle a request in Grails is to create a controller. Out of convenience, Grails provides a target to do just this. To create a controller, run the `create-controller` target and enter **hello** as the name of your controller followed by the Enter key. Listing 1-3 demonstrates this in action.

Listing 1-3. *Running the create-controller Target*

```
>grails create-controller
    [input] Enter controller name:
hello
```

This will create a new controller called `HelloController` within the `grails-app/controllers` directory of the Grails application as well as an associated unit test case called `grails-test/HelloTests.groovy`. As mentioned previously you could have created the controller with an IDE or text editor. Regardless, the resulting controller created for you will resemble something like this:

```
class HelloController {
      def index = { }
}
```

At the moment it contains only a single action called `index`. Yes, controllers have actions that they delegate to (another thing you'll learn about later in the book). Just to provide some clarity, the effect you want to achieve is depicted in the screenshot in Figure 1-3.

Figure 1-3. *The "Hello World!" response*

In order to create the response depicted in Figure 1-3, you're going to make some modifications to the `HelloController` and add a new action called `world` to it. Actions are both *closures* and *properties*. In the coming chapters you'll begin to understand more what this means at a practical level. Listing 1-4 demonstrates how to create the `world` action.

Listing 1-4. *The world Action*

```
class HelloController {
     // each action is a closure property
     def world = {
          render 'Hello World!' // render response
     }
}
```

The previous example defines a single action called `world` that renders the text "Hello World!" to the response. It does this using one of Grails' built-in methods called `render`, the details of which you'll learn about in Chapter 7. To try the "Hello World!" example, start the Grails application by running the following target:

```
grails run-app
```

Once the application has loaded, open a browser and navigate to the address shown in Figure 1-4 by typing the URL into the address bar (by default Grails starts up on port 8080).

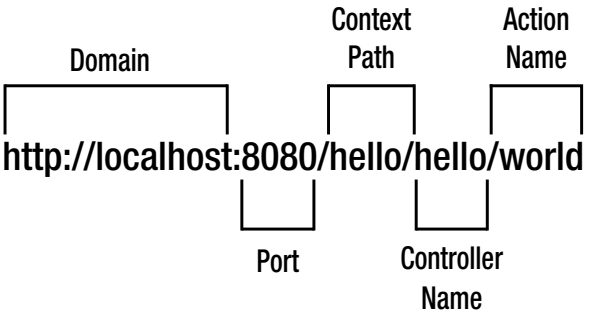

Figure 1-4. *The Grails URL format*

As the diagram illustrates, Grails uses a convention to automatically configure the path to a particular action. Following Figure 1-4, the *context path* is the root of the application and could be removed altogether if the final deployment environment is the root application. The *controller name* part of the URL is taken from the first part of the `HelloController` class's name. Essentially, the controller name is calculated as the controller class name minus the `Controller` suffix and the first letter in lowercase. Finally, the *action name* at the end of the URL shown in the figure maps to an *action* within the controller. In this case the URL will call the `world` action defined in Listing 1-4.

The result is that the text "Hello World!" will be displayed within the browser as depicted in Figure 1-5. Now, say instead of merely saying hello to the user you want to do something rather more useful such as displaying the current date and time.

To get the controller to display a message as well as the date and time, you need to open up the controller again and modify the `world` action as shown in Listing 1-5.

Listing 1-5. *Modified HelloController.groovy*

```
class HelloController {
     def world = {
       render "Hello World it's " + new java.util.Date()
    }
}
```

Figure 1-5. *"Hello World!" plus the date*

The `world` action in Listing 1-5 concatenates the string "Hello World!" with an instance of the class `java.util.Date`, the standard class for handling dates in the JDK. Once the modification has been made don't stop the server, simply hit the Refresh button on the browser and note how your changes are immediately available with the action printing the "Hello World!" text followed by the current date and time. No recompilation step, no build to run, and no packaging required. Grails automatically reloads the controller at run time without you, the developer, needing to restart the server.

This simple example demonstrates some of the key features of Grails, including a glimpse into its MVC architecture, the tight integration with Java (I used `java.util.Date`), and the iterative development made possible by autoreloading.

This is just the taster, however, and in future chapters we'll explore Grails with more concrete examples.

The Unit Tests

Yes, it seems nowadays that no code can be written without an associated test, and rightly so. Fortunately, Grails has already created a test case for us called `grails-test/HelloTests.groovy`. The initial code for this unit test is the following:

```
class HelloTests extends GroovyTestCase {
    void testSomething() {
        // test code goes here!
    }
}
```

As you can see there is a `testSomething` method just waiting for us to populate it. Nevertheless, there is an entire upcoming chapter dedicated to testing in Grails that tackles concepts such as basic unit testing, mock objects, and more. Hence, I'll defer going into any great detail about testing here until later. If you can't wait, the chapter in question is Chapter 6.

Summary

So far you learned how to create a Grails application, took a brief look at rendering responses with controllers, and had a glimpse at the Grails command-line targets. Clearly I only brushed the surface of Grails with the first example, and introducing the framework this way is little bit like plunging into the deep end of the Groovy and Grails world on Groovy.

In the next chapter I will go through an overview of the Groovy language and its key features, particularly those that are significant during Grails development. Feel free to skip Chapter 2 if you think you're already an expert on Groovy.

Chapter 2 is by no means a complete detail of the entire Groovy language, as that warrants a book by itself (and funny enough, there is one: Groovy in Action produced by Manning Publications), however it will give you a feel of the language and why it is such a great fit for web application development.