CHAPTER 1

An Introduction to Spring

The first time I encountered Spring was when a client asked me whether I knew anything about it. I didn't and said so, but that's always my cue to go find out about a technology. Next time, or so my reasoning goes, I should at least be able to reel off a definition.

Most of the documentation I could find stressed two basic points: that Spring supported inversion of control (IOC) and that it was a lightweight framework. I found this enormously puzzling because although the various sources discussed these features, none of them addressed the question of why these features were desirable.

The situation has improved somewhat since then. Most introductions to the Spring framework do make at least a gesture toward discussing the merits of the feature set rather than merely listing it. Even so, while this chapter is my chance to impart a respect for the technical accomplishments of the Spring authors, I also intend to explain just why some of those technical features are so valuable.

Two years after having to express total ignorance of Spring, I find myself using it every day because it allows me to build applications far more productively (and enjoyably) than I could have done before. I have found working with Spring to be enormously rewarding and I hope you will too.

Frameworks

I don't think there is any hard and fast definition of what does or does not constitute a framework. My rule of thumb definition would probably be that it's a *framework* if in general it invokes your code rather than your code invoking it—but there are plenty of self-professed frameworks that fall outside my rather narrow definition.

Certainly Spring is a framework by this definition. I discuss one aspect of this in the next section, "Inversion of Control," but this is not the only sense in which Spring could be said to be a framework, and indeed it is not compulsory for you to use Spring in this way; stand-alone applications can easily take advantage of various components of Spring.

A broader sense of *framework* defines it as a structure used to solve a complex technical issue. Again Spring qualifies, though it might be better to think of it as a framework of frameworks. For example, the Hibernate Object Relational Mapping (ORM) framework provides a solution to the complex technical problem of persisting objects from your object model into a relational database. Spring provides a set of tools to aid you in integrating Hibernate with the other parts of your applications—and Hibernate is only one of many frameworks and libraries that Spring provides support for.

Lightweight, another ill-defined term, can be taken as implying the lack of a need for a Java Platform, Enterprise Edition (Java EE) component stack, as the impact on your application's memory footprint, as the impact on your application's disk (and thus download) footprint, and as the degree to which you can discard unnecessary components. I do not think that the term *lightweight* has any real value, but in all of these areas Spring excels. Indeed, to a large extent it was created as a reaction against the weight of the Java EE component stack, though it is able to take advantage of Java EE features when this is desirable.

Spring is therefore an environment within which your code can operate, a set of libraries for solving certain types of problems, and a set of libraries for assisting you in interacting with numerous other frameworks. However you define the buzzwords, Spring is a fine example of a useful framework.

Inversion of Control (IOC)

A familiar problem to application developers is creating the application glue code—code that doesn't do much other than set up preexisting components and manage the data that is being passed between them. Typically, the problems arising from this concern exhibit themselves in monolithic brittle factory classes that become dependencies for large parts of the application and are virtually impossible to test in isolation.

At its heart, Spring is primarily a framework for enabling the use of IOC (also known as *dependency injection*). This is not to diminish Spring's other features, but rather to highlight the importance of IOC in addressing the problem of tangled dependencies. In this section, I will try to explain IOC's value.

Dependency Lookup

Typical application logic traditionally does something like the following to obtain a resource:

```
Foo foo = FooFactory.getInstance();
```

Here we have obtained the resource (an instance of `Foo`) by invoking a static method on a singleton factory class. Alternatively, we might construct the desired resource directly:

```
Foo foo = new FooImpl();
```

Or we might look up the resource in a Java Naming and Directory Interface (JNDI) context:

```
Context initialCtx = new InitialContext();
Context environmentCtx = (Context) initCtx.lookup("java:comp/env");
Foo foo = (Foo)environmentCtx.lookup("Foo");
```

I'm sure you can think of dozens of other ways that you can acquire resources, but most of them will have two things in common: your application logic is in control of exactly what resource is acquired, and you create a hard dependency on some other class in the process. This approach is known as *dependency lookup*.

In these three examples, we create dependencies upon the `FooFactory` class, upon the `FooImpl` implementation class, and upon the various classes of the JNDI application programming interface (API).

The Problem with Dependency Lookup

You could reasonably ask why dependency lookup is a bad thing. Obviously these techniques all have value. Certainly we aren't going to give up use of the `new` operator anytime soon. The disadvantage arises when we choose to reuse code that has a hard dependency on one set of classes in another context where they are less appropriate.

For example, consider some application code that acquires its database `Connection` object by use of the `DriverManager`'s factory methods, as is typical for a stand-alone application (see Listing 1-1).

Listing 1-1. *Acquiring Connection Resources by Using Factory Methods*

```
public void foo() {
   Class.forName("org.hsqldb.jdbcDriver");
   Connection c =
      DriverManager.getConnection("jdbc:hsqldb:timesheetDB","sa","");
   PreparedStatement ps =
      c.prepareStatement("...");
   ...
}
```

When we come to migrate this code into a web application where database resources are normally acquired by JNDI, we must modify the code. Ideally, we would keep all of the database connection acquisition logic in one place so that we need to change only one class, rather than changing all classes where the connection object is used. We can do this by providing a factory class, as shown in Listing 1-2.

Listing 1-2. *Simplifying Connection Acquisition by Using Another Factory*

```
public void foo() {
   Connection c = ConnectionFactory.getConnection();
   PreparedStatement ps =
      c.prepareStatement("...");
   ...
}
```

Alternatively we could do this by supplying the connection object to any classes that need to use it, as shown in Listing 1-3.

Listing 1-3. *Simplifying Connection Acquisition by Parameterization*

```
public FooFacility(final Connection c) {
   this.c = c;
}

private Connection c;

public void foo() {
   PreparedStatement ps =
      c.prepareStatement("...");
   ...
}
```

Of these two latter approaches, at first glance the `ConnectionFactory` class looks more appealing because it has a reduced footprint in our class. On the other hand, we still have a hard dependency on the external class. Our changes to ensure compatibility within different environments are certainly reduced—now we will have to amend only `ConnectionFactory`—but this class is still required, and in environments where there is already a strategy for connection acquisition, it will add complexity to add another class with the same responsibility. You would naturally want to replace calls to our custom `ConnectionFactory` with calls to the existing factory (or vice versa), but this brings us back to our original problem: having to modify code when moving our logic to a new environment.

Dependency Injection as a Solution

If we use the parameterized version of the code, we have removed the need to modify the code in any environment because we have removed the hard relationship with the classes that create the `Connection` object. To use the correct terminology, we have *decoupled* our class from the *dependency* required to appropriate the connection.

The problem with decoupling the logic in this way is that it potentially creates a tedious requirement to provide the connection whenever we wish to use this logic. Using the appropriate terminology, this is the problem of how to *inject* the dependency. This is exactly the problem that Spring IOC solves: it makes the problem of supplying dependencies to classes so wonderfully simple that we can take full advantage of the benefits of decoupling.

I explain in detail how you inject dependencies by using Spring and how this mechanism works internally in Chapter 3.

Dependency Injection as an Aid to Testing

I have explained how tight coupling causes problems when we want to move our application logic from one environment to another, but there is a special case of this issue that makes IOC's advantages dramatically apparent. This is the problem of unit testing.

Writing unit tests is an art in itself. The well-written test concentrates on a single component of the system and tests all of its behavior as thoroughly as possible. However, when a class is tightly coupled to other parts of the application, testing that class in isolation becomes impossible.

By encouraging loose coupling, it becomes easier to eliminate irrelevant classes from the test, often by providing mock objects in place of heavyweight implementations. I discuss unit and integration testing in more detail in Chapter 10.

An Agile Framework

A variety of successful software development techniques have become known collectively as *agile programming*. Initially having a very loose definition, agile development became codified in the "Agile Manifesto" (`www.agilemanifesto.org`) presented by a number of software development luminaries.

There are now several formal methodologies such as Scrum and Extreme Programming (XP) that follow agile approaches. The precise value of the full collection of techniques used is debatable, and some shops that pay lip service to agile methodologies don't follow through on all of the agile edicts. Nonetheless, the agile approach is becoming ever more popular, and even in isolation the individual techniques of agile programming are certainly proving their worth. The need to issue frequent deliverables and the encouragement of refactoring present challenges to traditional environments, where tight coupling between components makes for difficulty in achieving the rapid rate of change that Spring can accommodate easily. Spring is not in and of itself an *agile* framework (there's no such thing) but it does lend support to some of the agile development techniques in various ways.

The ease of testing a cleanly decoupled Spring application accommodates the Test-Driven Development (TDD) approach. Spring espouses the Don't Repeat Yourself (DRY) principle. This encourages developers to create logic once and only once. Wherever possible, boilerplate code has been abstracted away into standard helper and library classes, and there are utility classes available to encourage developers to adopt the same approach for their own implementations. Code constructed in accordance with the DRY principle makes refactoring easier, as changes to application logic are localized.

Aspect-Oriented Programming (AOP)

Aspect-oriented programming (AOP) describes an approach to a set of problems that do not fit naturally into the object-oriented programming (OOP) approach to problem solving. AOP is not particularly new, but in the Java world it is only with the introduction of tools such as AspectJ and Spring AOP that it has gained a mainstream audience.

Unfortunately, AOP introduces its own terminology, which Spring AOP has adopted for the sake of consistency with existing tools. The concepts are remarkably simple, however, even when the underlying implementation is complex.

The use of an AOP framework allows a solution to a problem to be applied before and after the invocation of various externally identified method calls. This is a gross approximation to the depth of AOP, which I discuss in far more detail in Chapter 5, but it should be sufficient for this introductory chapter.

Almost all Spring developers will want to take advantage of existing AOP libraries in Spring to apply to their own applications. The most typical example of this is the declarative transaction management library. In a conventional Java application, a service layer method's transaction management might be handled something like this:

```
public class AccountServiceImpl
   extends ServiceImpl
   implements AccountService
{
   public Account createAccount() {
      try {
         beginTransaction();
         Account account = dao.save(new Account());
         commitTransaction();
         return account;
      } catch( Exception e ) {
         rollbackTransaction();
      }
   }
}
```

With the use of declarative transaction management, the method implementation can be reduced to this:

```
@Transactional
public class AccountServiceImpl
   extends ServiceImpl
   implements AccountService
{
   public Account createAccount() {
      return dao.save(new Account());
   }
}
```

Instead of duplicating the begin/commit/rollback logic in all of our service layer implementation classes, we use a Spring AOP annotation to declare that a transaction must begin when we enter any of the implementation class's methods and that it should be committed when they complete. We also accept the default behavior that causes unchecked exceptions emitted by the method to roll back the transaction. The syntax of all this is remarkably compact.

Because Spring provides all of the AOP libraries necessary to carry out the transactional behavior identified by our annotation, no further configuration is required.

Note I think this is a big enough deal that it's worth reiterating: a tiny annotation removes the need for any explicit transaction management anywhere else in your application.

AOP can be applied anywhere that you have a set of requirements that apply without regard to the object model across otherwise unrelated parts of your application. Indeed, functionality that addresses these concerns is essentially the definition of an *aspect*. The commonest uses of AOP are therefore in managing transactions, guaranteeing security, and providing auditing and logging information. These are all supported by existing Spring AOP libraries, to such an extent that typical Spring developers will never need to create their own AOP libraries. Even so, Chapter 5 covers the creation of simple AOP tools along with the alternative XML-based syntax and use of the AspectJ framework.

Libraries

Spring doesn't just provide a bare framework and leave other libraries to their own concerns. Instead it provides wrappers to accommodate other design philosophies within its own framework.

All of the standard parts of Java EE are supported. You can therefore manage JTA transactions, connect to databases, enforce security, send e-mail, schedule operations, manage JMX services, generate reports, write PDF files, and in fact do pretty much anything you are likely to want to do.

For the rare case that falls outside Spring's coverage, Spring is emphatically based around the use of Plain Old Java Objects (POJOs) and allows for the initialization of almost any preexisting class that can be invoked from conventional code. It is trivially easy to integrate even the most cumbersome of legacy code.

In practice, the Spring philosophy is so alluring that developers familiar with Spring are likely to add wrappers (again a variety of classes exist to assist with this) to existing code to give it a more Spring-like external appearance—when they can resist the temptation to rework the internals of the offending library.

Spring and Web Applications

In some ways, Spring was created both as an attempt to sidestep the overbearing requirements of Java EE and also to gain some of its advantages. The problem with Java EE historically was that although it provided a lot of excellent features, it was difficult to use these in isolation, forcing developers to choose between the heavyweight complex Java EE environment and simpler but limited alternatives. Spring bridges this gap by allowing developers to pick and choose the most appropriate parts of Java EE for their applications. It applies this approach to a variety of other libraries and toolkits, and adopts the same philosophy to its own internal design.

Java EE is and was primarily a platform for server programming. Spring can be used entirely independently of the server environment, but it provides strong support for server programming and particularly for web application building.

Spring MVC

My commercial exposure to the Spring framework in general arose through a specific requirement that we use the Spring Model View Controller (Spring MVC) framework to build the web component of an application, so I have something of a soft spot for it.

A Spring MVC application is cleanly divided between views, controller classes, and the model. The views are typically JSPs, though they can use a variety of other technologies. A suite of controller classes are provided that cover everything from the creation of basic forms to fully fledged “wizard” classes that allow you to walk a user through filling in a complex form. The implementation of the model is up to you, but typically consists of a service layer in turn calling into data access objects (DAOs) for persistence requirements.

As with all good frameworks, Spring MVC does not force you to use session scope to maintain state (ensuring good scalability). While the controllers take advantage of inheritance to provide most of their functionality, it is trivially easy to implement a controller interface and aggregate in existing controller behavior, allowing your own controller classes the option to aggregate or inherit external functionality. Most other web frameworks are not as liberal.

The transfer classes (form beans) in Spring MVC are conventional POJOs, and the validation framework is both POJO-based and simple to configure.

Spring Web Flow

Spring Web Flow can be seen as a complement to the existing Spring MVC framework, as it uses the existing view resolvers and a specialized controller to provide its functionality. Web Flow allows you to model your application behavior as a state machine: the application resides in various states, and events are raised to move the application between these states. That may sound a bit weird if you haven't seen this sort of model before, but it's actually a pretty well-accepted approach to designing certain types of web applications.

Web Flow allows you to design modules of your web application as complex user journeys without arbitrary end points. Whereas Spring MVC is ideal for simple linear form-based problems, Spring Web Flow is suited to more-dynamic problems. The two can be mixed and matched as appropriate.

The additional advantage of building an application by using Web Flow is the ease of design—state machines are easy to model as diagrams—combined with the fact that a Web Flow application can readily be packaged for reuse in other projects.

The web component of our example application is built using a combination of Spring MVC and Spring Web Flow so you will have an opportunity to gauge the relative merits of these two related approaches to web application design.

Spring Portlet MVC

Of specialized interest to Portlet developers is the Spring Portlet MVC framework. Portlet containers (portals) allow you to build a larger web application up from a set of smaller subcomponents that can reside together on the same web page. Portals usually provide a set of standard infrastructure capabilities such as user authentication and authorization. A typical portal is supplied with a large suite of standard portlets to allow users to read e-mail, manage content, maintain a calendar, and so on. This makes them attractive for creating in intranets or for customer-facing websites, where a set of basic services can be supplemented by a small suite of custom-written tools to provide an integrated environment without the expense and time constraint of creating an entirely bespoke system.

Spring Portlet MVC provides an exactly analogous version of the Spring MVC framework for working within a JSR 168–compliant portlet environment. Although Spring Portlet MVC builds on the JSR 168 portlet API, the differences between Spring Portlet MVC and Spring MVC are much easier to accommodate than the differences between the portlet API and the servlet API that underlie them.

In addition to minimizing the technical differences between the portlet and servlet APIs, Spring Portlet MVC provides all of the facilities to the portlet environment that Spring MVC provides to the servlet environment.

Other Frameworks

While it introduces some delightful frameworks of its own, Spring also plays nicely with existing frameworks. There is full support for the use of Apache Struts, JavaServer Faces, and Apache Tapestry in the framework. In each case, suitable classes are provided to allow you to inject dependencies into the standard implementation classes.

Where possible, several approaches are offered for users who may be working under additional constraints. For example, the Struts framework can be Spring enabled by configuring your actions using either `DelegatingRequestProcessor` or `DelegatingActionProxy`. The former allows closer integration with Spring, but the latter allows you to take advantage of Spring features without giving up any custom request processors that you may be using (Struts does not allow you to configure multiple request processors).

Similar support is available for most commonly used frameworks, and the approaches used for these transfer well to any other web framework that uses standard Java features and that provides for a modicum of extensibility.

Other Issues

A typical example of Spring's helpfully catholic perspective is in its support for creating DAO classes. Spring provides a common conceptual approach of template and helper classes that you will examine in more detail in Chapter 4. Specific classes are provided for the various database persistence tools, including plain JDBC, but also ORM tools such as Hibernate, iBATIS, and TopLink.

Security is addressed by the Acegi Spring Security component. This provides a comprehensive suite of tools for enforcing authentication and authorization in a web application. I discuss the Spring Security framework in Chapter 7.

Spring has a wealth of other features that are not specific to any one framework, but which are enormously helpful. There is support for a suite of view technologies, including traditional JSPs but also encompassing XML, PDF files, Apache Velocity, and even Microsoft Excel spreadsheets.

Support for features such as the Jakarta Commons file-upload utilities and the notoriously tricky JavaMail API turn otherwise problematic tasks into relatively simple configuration options.

Documentation

Documentation does not normally appear on the feature list of any framework, and open source tools have a mediocre reputation for their documentation. Typically, developers are more interested in writing interesting software than in explaining to the uninitiated how to take advantage of it. Spring is a breath of fresh air in this respect. The documentation for Spring itself is well written and comprehensive.

The Spring Javadoc API documentation is particularly well thought out, another happy surprise for developers too used to seeing the minimum of autogenerated API references. For example, the Javadoc for the Spring MVC framework discusses the purpose of the various classes, methods, fields, and parameters in depth, but it also contains invaluable discussion of the life cycle of the controller classes.

Spring is a formidable product, without doubt. Because it ties together such a diverse suite of libraries and other frameworks, it inevitably has some murky corners and contains some pitfalls for unwary novices. This book aims to address those issues and help you up the steeper part of Spring's learning curve. After you have bootstrapped a basic understanding of the design and philosophy of Spring, you will have a wealth of documentation and other resources available to you.

All of the documentation for the Spring framework is available from the Spring website at `http://springframework.org`, and you can get help from a thriving community of other Spring users in the forums at `http://forum.springframework.org`.

Other Tools

While Spring is primarily a set of libraries constituting a framework, I should mention the tools typically used when working with Spring, and the support that is available for them.

Maven

Spring does not require specific support from its build environment. Still, Spring's broad spectrum of support for external libraries can lure a developer into creating a project that has a complicated dependency tree. I would therefore recommend the use of a tool providing support for dependency management. For the examples in this book, I have used the Maven 2 project to manage dependencies, and it is gratifying to note that the files in the default Maven repository are well maintained for the Spring framework. Users of other dependency management tools that take advantage of Maven repositories will also benefit from this good housekeeping.

Spring Integrated Development Environment (IDE) Plug-in

Spring uses XML files for its configuration, and all current integrated development environments (IDEs) will provide basic support for maintaining correct XML syntax. Most IDEs now also provide a modicum of additional support for Spring configuration files.

The examples in this book were all built using the Java Development Tools edition of the Eclipse IDE. Eclipse does not provide innate support for Spring beyond its XML capabilities, but it is trivial to install the Spring IDE plug-in for Eclipse. This provides intelligent sensing of the attributes in bean configuration files, and a wizard for creating the contents of a basic Spring 2 project. I provide a walk-through of the basic features of this plug-in in the appendix.

Conclusion

Spring is more than the sum of its parts. Although some of the subcomponents of Spring are important projects in their own right (Spring MVC is the example that springs to mind), Spring's major contribution is that it presents a unifying concept. Spring has a definite philosophy of design, and wrappers are provided for libraries that deviate from this philosophy. To a large extent, when you have learned to use one library within the Spring API, you will have equipped yourself with a large part of the mental toolkit that is required to use all the others.

Rather than worrying about the time it will take to use a new technology, Spring developers for the most part can be confident that they will know how to configure and interact with the tools it comprises. The freedom to integrate tools into an application without the fear of spiraling complexity encourages us away from the tyranny of Not Invented Here syndrome. In short: Spring makes you more productive.

In the next chapter, you'll look at the sample application that we'll be using to illustrate the use of the Spring framework as a whole, and then in subsequent chapters I'll take you through the individual features and show you how they are used to build the application.