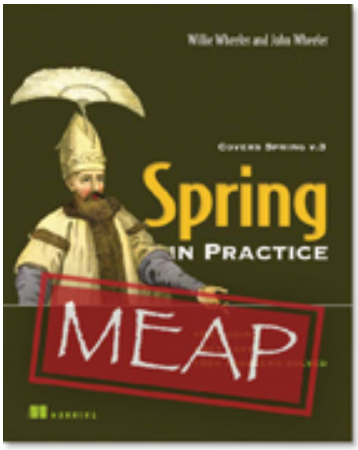
MANNING PUBLICATIONS



Spring in Practice
By Willie Wheeler and John Wheeler

The Spring Framework is a container environment and complementary set of APIs that aim to simplify the development of enterprise Java software. At the framework's core lies a dependency injection container that allows us to specify components and their dependencies, often in a highly automated manner. In this green paper based on Spring in Practice, the authors give a brief rundown of each the components that make up the framework and specifically of dependency injection.

To save 35% on your next purchase use Promotional Code **wheelergp35** when you check out at http://www.manning.com/.

You may also be interested in…

Introducing Spring

Hello, and welcome to Spring in Practice! In this green paper, we'll give a brief rundown of each the components that make up the framework, but since dependency injection is such an important, fundamental concept, we'll focus primarily on that. We'll provide a little upfront theory by examining a concept called *inversion of control* and how it relates to dependency injection.

Most of the objects we configure with Spring are architectural components as opposed to domain objects. Common configuration targets include, for instance, infrastructure (JDBC data sources, JavaMail sessions, and so forth), data access objects, service beans and MVC controllers. Spring is flexible and allows us to configure such components and their interdependencies using XML, Java 5 annotations, or even Java itself.[1]

What is Spring and why use it?

The Spring Framework is a container environment and complementary set of APIs that aim to simplify the development of enterprise Java software. At the framework's core lies a dependency injection container that allows us to specify components and their dependencies, often in a highly automated manner. It's easy, for example, to make the container discover components for us (we can provide base packages for the container to scan), and it's even easy to make the container discover the dependencies. But, if we prefer to be more explicit about our component and dependency definitions—maybe we see the configuration as an executable piece of architectural documentation—that's of course an option for us as well. It's entirely up to us.

The container is feature-rich—sufficiently rich that it's not possible to give it a full treatment in a single chapter. But we're not aiming to be comprehensive. Instead, our aim is to equip you with the knowledge and techniques you'll need to understand the rest of this book. We're going to highlight and explain some best practices around using the dependency injection container, and we're going to show how to tackle real problems that we think are representative of the type of work that developers are asked to do every day.

[1] See http://www.springsource.org/javaconfig for information on the Java Configuration project.

The major pieces of the framework

Figure 1 shows a high-level block diagram of the parts we're going to discuss.[2]

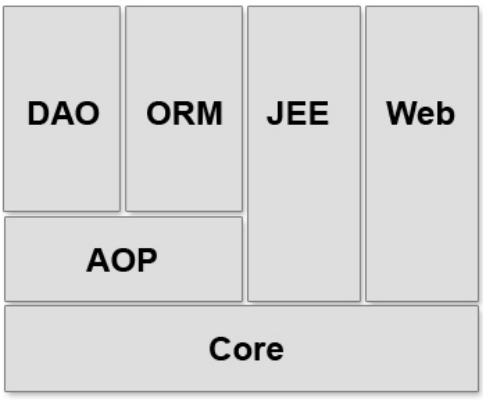


Figure 1 High-level diagram of Spring's major components

THE CORE DEPENDENCY INJECTION CONTAINER

The dependency injection container is a core feature of the Spring Framework. We will further dissect what DI is in the next section, but for now, it's enough to know that it enables complex object configuration and initialization. Sophisticated object graphs and their constituent objects' configuration parameters (for instance, database connection strings, web service endpoints, and usernames and passwords) can be retrieved from the container via a single method call. The container also provides first-class support for internationalization (i18n) and seamless integration with different programming environments such as servlet and portlet containers.

ASPECT-ORIENTED PROGRAMMING

The Spring Framework also supports aspect-oriented programming with both a simpler approach called Spring AOP and the more powerful AspectJ approach. Like the DI container, AOP support is both independently useful to developers and is used to implement different parts of framework functionality. For example, Spring implements its support for declarative transaction management through AOP since transactions are a cross-cutting concern.

DATA ACCESS OBJECT SUPPORT

Spring's data access object (DAO) support relieves developers from having to write error-prone template code (whether that's JDBC boilerplate, Hibernate boilerplate, or something else) by automatically managing database connections and connection pools, and by mapping vendor-specific errors into a uniform exception hierarchy. It also makes it easy to map `java.sql.ResultSets` to lists of domain objects and execute stored procedures.

OBJECT-RELATIONAL MAPPING SUPPORT

If you prefer to use object-relational mapping (ORM) instead of straight JDBC for database access code, you're in luck. The Spring Framework supports the best and most popular ORMs available including Hibernate, JPA, iBatis, JDO, and Toplink. The declarative transaction management mentioned previously also works with this ORM support.

ENTERPRISE JAVA SUPPORT

Spring provides abstractions that make working with different enterprise Java technologies easier. For example, an abstraction over the JavaMail API, Spring's `JavaMailSender`, makes sending emails more straightforward (even those with attachments). There is also an abstraction over JMS, `JmsTemplate`, that makes sending and receiving messages easier. Abstractions and integration points exist for JMX, remoting and web services, scheduling, and more. The framework tracks JEE closely and generally supports JEE specifications as they become available.

---

[2] This graphic was inspired by one in the official Spring reference documentation here: http://static.springframework.org/spring/docs/2.5.x/reference/introduction.html

WEB SUPPORT

Last in the framework stack is Spring's web support. Not only does it integrate with popular web development frameworks and technologies like Struts, JSF, Velocity, Freemarker and JSP, it provides its own powerful framework: Spring Web MVC. Since many of the web frameworks it integrates with as well as its own are MVC-based, Spring also supports Excel spreadsheet and PDF views. In addition, it has integration points with file uploading APIs like Jason Hunter's `com.oreilly.servlet`.

So, why use it?

You may have worked with or even developed other frameworks or APIs that handle one or more of the Spring Framework's concerns. So why would you stop to learn something that requires a fairly substantial time investment? There are several good reasons.

- **Popularity.** First, Spring is a very popular choice, as evidenced by the myriad publications, websites, and job postings that reference Spring in some fashion. While popularity isn't necessarily in and of itself a reason to adopt something, it's better to adopt frameworks that the community embraces and supports. And usually that popularity has a basis in the quality of the framework itself.
- **Quality.** That leads us to our second reason: the framework really is quite powerful and useful with respect to its stated goal of simplifying JEE development. Its designers and developers are well-known industry experts in JEE development. They've worked hard to factor out repetitive code, simplify difficult APIs, and provide integration points with best-of-breed technologies. And they've done a nice job with it.
- **Promotes best practices.** Spring promotes best practices. For example, when using JDBC, we're not only responsible for writing SQL and mapping `java.sql.ResultSets` to domain objects; we're also responsible for managing database connections. Spring makes a non-issue out of this through `JdbcTemplate`, which uses the template pattern to separate JDBC boilerplate from the "good stuff" that varies from case to case. Also, there isn't any good reason transaction code should be duplicated throughout a project, but in many projects there isn't a decent, readily-available way to centralize it. With Spring, we can declare that certain methods participate in transactions using either XML or annotations, and never write a lick of Java transaction code ourselves. Finally, Spring is non-invasive, meaning that it stays out of your way as much as it can. For example, Spring Web MVC controllers and forms can be ordinary POJOs. While often times we have to annotate these POJOs to enable advanced functionality, the fact that they don't extend and depend on Spring APIs directly can keep them useful in other contexts.
- **Modularity.** Spring is modular. It isn't an all-or-nothing solution. We can pick and choose what's of interest and disregard what we don't need. Spring has been designed from the ground up with this in mind. The codebase itself doesn't have any cyclic dependencies[3]. Moreover, the distribution ships with components as individual jars that can be included piecemeal in a project as well as a single jar that includes the entire set of components.
- **Modest learning curve.** Finally, Spring really isn't *that* hard to learn. For example, once we pick up `JdbcTemplate`, we're on our way to understanding `HibernateTemplate`, `SqlMapTemplate` (for iBatis) or even `JmsTemplate`. We'll find as we make our way through the framework, common patterns emerge. Plus there are hundreds of resources online and in print at our disposal including message boards where the core developers often participate.

For an excellent print reference that will certainly be of aid during your journey with this book, check out Craig Wall's *Spring in Action*, Second Edition (Manning Publications).

Spring offers a lot, and no doubt it takes time to understand and appreciate the landscape. Rest assured, however, that the effort is well worth it. By learning Spring and using it to solve problems, we'll see how to bring together disparate technologies and incorporate them into a cohesive applications. We'll keep hardcoded configuration parameters out of our classes and centralized in standard locations. We'll design interface-based dependencies between classes and better support changing requirements. And, ultimately, we'll get more done

[3] You can read an article by Keith Donald, a Spring Framework core contributor, about how the Spring Framework doesn't have any cyclic dependencies here: http://www.springframework.org/node/310

with less effort and in less time because the Spring Framework handles the plumbing while we focus on writing code to solve business problems.

Now, that we have a general idea of what the framework offers, it's time to take a look at arguably the most important concept that will carry us forward: inversion of control via dependency injection.

Flexible configuration via dependency injection

Inversion of control (IoC) became popular some years back through dependency injection (DI) containers like Spring, so while that might be eons ago in Internet time, it is still a relatively new and unfamiliar concept for many developers. In this section, we'll explain what IoC is and examine the forces that produced it. We'll even get our hands a little dirty and see how to configure Spring's container.

Configuring dependencies the old way

In the beginning, we learned we could model the real world and its business systems with object-oriented programming languages. This is all good and true, but it isn't the whole story. Practically speaking, the domain objects we build aren't very useful without their surrounding context, and the business models we build need user interfaces. We keep business objects isolated from the code that makes them persistent and we make sure data access code doesn't bleed into presentation code. We layer our systems to enforce a separation of concerns, which sounds easy on paper but is often harder in practice.

Consider the relationship in between a data access object and the `DataSource` it relies upon. For the DAO to work with the `DataSource`, we need to initialize the `DataSource` with various connection parameters.

```
public class JdbcAccountDao implements AccountDao {

    private BasicDataSource dataSource;

    public JdbcAccountDao() {
        dataSource = new BasicDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl(
            "jdbc:mysql://localhost:3306/springbook?autoReconnect=true");
        dataSource.setUsername("someusername");
        dataSource.setPassword("somepassword");
    }
    ...
}
```

An obvious problem here is that the connection parameters are hardcoded. It's likely that many DAOs will share this connection information, so repeating it across multiple DAOs is going to lead to multiple code changes, recompilations and redeployments every time a parameter changes.

We could externalize the connection parameters with `java.util.Properties`, and that would certainly be an improvement. But a more subtle problem would remain.

Figure 2 below shows the dependency we've introduced between `JdbcAccountDao` and Apache DBCP's `BasicDataSource`, a `DataSource` implementation. We know that we're going to need a `DataSource`, and we don't mind the interface dependency; in fact, we want the dependency strictly on the interface to keep things flexible in case better options surface. Even with an interface dependency, we're still going to have to initialize a concrete class and then cast the reference because the interface doesn't specify the parameter setting methods we're calling. In other words, we'll still have the concrete dependency.

Figure 2 JdbcAccountDao has a dependency on BasicDataSource, which is a concrete DataSource implementation.

Dependency injection

One way to eliminate the concrete dependency from `JdbcAccountDao` would be to create the dependency externally and inject it in `JdbcAccountDao`. This gives us a lot of flexibility because we can easily change the configuration in one place, which means that it's easier to do it more often, should we want to do so. If we want to proxy the `DataSource` before injecting it, we can do that. In unit testing scenarios, if we want to replace the `DataSource` with a mock, we can do that too. Again, dependency injection provides a lot of flexibility that we don't have when the dependency's construction is hardwired into the components relying on the dependency.

To make DI work, we need create the `DataSource` externally, and then either construct the DAO with it or set it on the DAO with a setter method, as we show below.

```
public class JdbcAccountDao implements AccountDao {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    ...
}
```

Our DAO no longer has a hardwired dependency on a particular DataSource configuration, and that obviously represents an improvement. But one might argue reasonably that we've succeeded only in pushing the construction upward into client code.

```
public AccountService() {
    try {
        Properties props = new Properties();
        props.load(new FileInputStream("dataSource.properties"));

        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName(
            props.getProperty("driverClassName"));
        dataSource.setUrl(props.getProperty("url"));
        dataSource.setUsername(props.getProperty("username"));
        dataSource.setPassword(props.getProperty("password"));

        this.accountDao = new JdbcAccountDao();
        this.accountDao.setDataSource(dataSource);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Actually, in one respect, we've made things even worse: we've introduced a dependency between `AccountService` and `BasicDataSource`—a relationship that is clearly undesirable. Furthermore, we also have a dependency between `AccountService` and `JdbcAccountDao` (a concrete class), so we're still in the same boat we started out in (see figure 3 below)! It's easy to see how the entire dependency graph for a particular system could become complicated and inflexible with nodes that are hard to swap out.

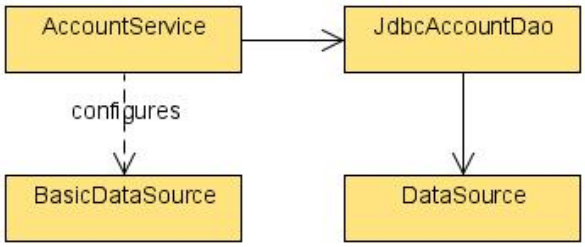


Figure 3 Now JdbcAccountDao has the desired interface-dependency on DataSource, but AccountService has dependencies on two concrete classes.

That doesn't mean that DI was a failed experiment. It's taking us in the right direction. To clean things up, we just need to revise *what* is doing the injecting.

Inversion of control

We can move the DI away from client code and over to Spring. In this scenario, client code doesn't request or look up an `AccountService`. Instead, the `AccountService` is transparently injected into client code when the client code is initialized. The code below shows `AccountService` with a strict interface dependency on `AccountDao`.

```
... imports omitted...

public class AccountService {

    private AccountDao accountDao;

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
}
```

So, how do we specify the dependency chain? With Spring, one option is to use XML to assemble it declaratively as in listing 1.

Listing 1 A spring configuration file that specifies object relationships

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">                                          #1
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url"
value="jdbc:mysql://localhost:3306/springbook?autoReconnect=true"/>
        <property name="username" value="someusername"/>
        <property name="password" value="somepassword"/>
    </bean>

    <bean id="accountDao"
        class="springinpractice.ch1.dao.jdbc.JdbcAccountDao">
        <property name="dataSource" ref="dataSource"/>                   #2
    </bean>

    <bean id="accountService"
        class="springinpractice.ch1.service.AccountService">
        <property name="accountDao" ref="accountDao"/>                   #3
    </bean>
</beans>
```

#1 DataSource configured with parameters
#2 DataSource injected into JdbcAccountDao
#3 JdbcAccountDao injected into AccountService

If you're new to Spring, the configuration above might be unfamiliar, but its meaning should be clear enough. At `#1`, we declare our `DataSource` and set it up with its configuration parameters. At `#2`, we declare our `JdbcAccountDao` and inject it with the `DataSource`. Similarly we inject the `JdbcAccountDao` into the `AccountService` at `#3`. Finally, the end result is that the service now carries the entire dependency chain, and the configuration is entirely transparent to the service. The cleaned up, new relationship is shown in the class diagram in figure 4.

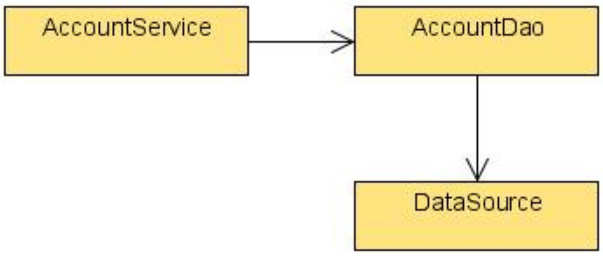


Figure 4 Now the dependencies are interface based and the concrete classes are configured transparently through Spring.s

Summary

This green paper has been an overview of some of the characteristics and benefits of the Spring DI container. Spring is a container that makes it easier to define a dependency graph for the components of your application.

Here are some other Manning titles you might be interested in:

Spring in Action, Third Edition
Craig Walls

Spring Batch in Action
Thierry Templier, Arnaud Cogoluegnes, Gary Gregory, and Olivier Bazoud

Spring Integration in Action
Mark Fisher, Jonas Partner, Marius Bogoevici, and Iwein Fuld

Last updated: November 10, 2011