Expert Spring MVC and Web Flow

Seth Ladd
with Darren Davison,
Steven Devijver and Colin Yates

Apress®

Expert Spring MVC and Web Flow

ISBN-13 (pbk): 978-1-59059-584-8

ISBN-10 (pbk): 1-59059-584-X

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewers: Rob Harrop, Keith Donald
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editor: Stephanie Provines
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor and Artist: Van Winkle Design Group
Proofreader: Nancy Sixsmith
Indexer: Broccoli Information Management
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 3

Spring MVC Application Architecture

Before we begin our exploration of the internals of Spring MVC, it is important to discuss how a typical Spring MVC is built. In this chapter, we will answer such questions as, "Where should the business logic live?" and "What are the correct levels of abstractions?" We will present the entire picture of a Spring MVC web application to better explain the individual roles that Spring MVC plays and where it fits in the overall architecture.

Mastering a new framework requires more than studying its APIs. Examining and understanding the architecture of a complete Spring MVC web application provides you with clues and motivations for the design of the framework itself. This allows for a higher level of understanding, allowing you to make more contextually sound choices when building your application.

Layers of Abstractions

Spring MVC applications are broken down into a series of layers. We consider a *layer* to be a discrete, orthogonal area of concern within an application. For instance, all of the persistence code is considered a separate layer from the view rendering code. Layers are abstractions within an application, and interfaces provide the contract by which layers interact. Some layers might be well hidden, used only by the layer immediately above it. In contrast, the most important layer (the domain model itself) spans nearly all the other layers in the system.

Layers are conceptual boundaries and are not necessarily physically isolated. More often than not, all of the layers will be located within the same virtual machine for a web application. For a good discussion on application distribution, consult Rod Johnson's *Expert One-on-One J2EE Design and Development* (Wrox, 2002).

Note Are layers the same thing as tiers? Many people use the two terms interchangeably, but separating the two helps when discussing the application and its deployment. A *layer* is a logical abstraction within an application. A *tier* is best thought of as a physical deployment of the layers. Thinking in layers can help the software developer, while thinking in tiers can assist the system administrator. Layers are mapped onto tiers.

Thinking in layers can help conceptualize the flow through an application. Visualizing the application's layers as a cake (layers of cake stacked one on another) is a common and convenient way to illustrate how the application is organized. Typical metaphors such as "down into persistence" and "back up to the user interface" refer to a cake, and denote a sense of vertical direction, reinforcing the metaphor. Figure 3-1 illustrates the common, highly generalized layers for web applications.

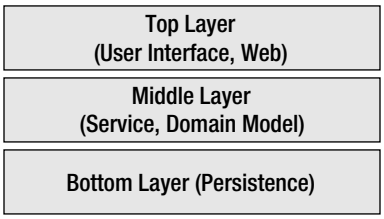

Figure 3-1. *General, high-level layers in a web application*

Typically, any persistence functionality is at the bottom of the cake, while the user interface is at the top. What is found in the middle, and how it is organized, is the subject of this chapter. We will use this metaphor when explaining our architecture.

Breaking it down further, typical Spring MVC applications have at least five layers of abstraction that you as a developer will code to. The layers are

- user interface
- web
- service
- domain object model
- persistence

You might notice that common applications elements, such as transaction management or security, are not in the preceding list. If you are familiar with the Spring Framework and its extensive use of aspect-oriented programming (AOP), this won't come as a surprise. Transaction management, for instance, is considered a transparent aspect of a system, not a full layer.

Figure 3-2 more specifically illustrates the relative placement of the different layers.

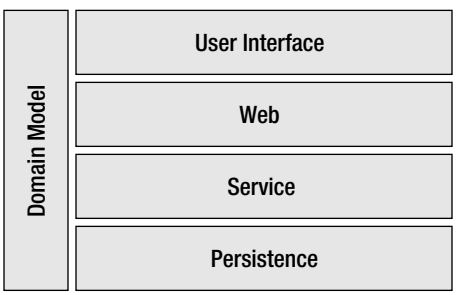

Figure 3-2. *Spring MVC application layers*

You will notice that the domain model vertically spans all the other layers. This is because all the other layers have a dependency on the domain model. It is the only layer that crosscuts all the rest.

Layer Isolation

Isolating problem domains, such as persistence, web navigation, and user interface, into separate layers creates a flexible and testable application. Implementations of each layer will vary independently, increasing the flexibility of the application. Decreasing the coupling between areas of the application will increase the testability, making it easier to test each part of the application in isolation.

This isolation is accomplished by minimizing the amounts of dependencies between the layers. The fewer dependencies a layer has upon itself, the less costly it is to change that layer. It is a best practice to ensure that a layer is required only by one or two other layers. Avoid having one single layer required by many different parts of the application.[1]

You can avoid dependency explosion in at least two ways. If a layer begins to employ too many layers, consider creating a new layer of abstraction wrapping all the previous interactions. On the other hand, if you find that a layer has permeated throughout many layers, consider if that layer is itself an aspect of the system. If the functionality can be applied across great swaths of the system transparently, use Spring's AOP functionality to remove the explicit dependency from your code.

The important point to remember is that one of the great benefits of layering an application is it creates a decoupled design. When you discover a layer or facet of the application that is too intrusive, refactor it to another abstraction or through AOP. This will keep your application flexible and testable.

Java Interface As Layer Contract

The Java interface is the key enabler to building an application with layers. The interface is a contract for a layer, making it easy to keep implementations and their details hidden while enforcing correct layer usage.

The benefits of low coupling provided by interfaces have been well known for some time. Their full benefits have been hindered because the instantiation of concrete types was still required. The promise of implementation abstraction wasn't quite realized—this is, before Spring and other Dependency Injection frameworks. Spring helps interfaces truly shine, because it handles the creation of the objects instead of your application code.

Treating an interface as a contract between layers is very helpful in large team settings. Coordinating the many resources often required by large projects is difficult, and it is rare that integration between layers happens precisely. Interfaces help to speed development between teams because of their lightweight nature. Developers program against the interface, while its implementation continues to be built and tested.

On a practical level, the Spring Framework works especially well with interfaces. Its AOP facilities are built around JDK proxies,[2] making it easy to extend an implementation of an interface with additional services.

---

1. Exception to this rule is the domain model, which typically spans many layers.
2. Spring can weave aspects into classes without interfaces, but a few caveats are involved. For one, the use of cglib (`http://cglib.sourceforge.net`) is required.

Using interfaces to define interactions between components makes unit testing much simpler. By using frameworks such as EasyMock (`http://www.easymock.org`) or jMock (`http://www.jmock.org`) you can easily create mock implementations of your interfaces with just a few lines of code and use these mocks when testing an object and its interactions with its collaborators. Although both EasyMock and jMock can create mocks of classes rather than interfaces, they require runtime bytecode generation and have some caveats that are not present with interface-based mocks.

■ **Note** We will provide extensive coverage of testing a Spring MVC application later in the book. You will learn how to use a mock objects library to help test individual layers of your application.

Connecting layers with interfaces also has an added benefit of reducing compile times and creating more modular builds. Concrete implementation classes can now change without requiring a recompile of any clients dependent on it (because the clients don't have a physical dependency on any concrete classes). For large systems, this can be very helpful at build and deploy time.

Using interfaces also enables systems to be so flexible that their implementations can be chosen at startup, or even while the application is runtime. Because the client is compiled against the interface, the implementation class can be swapped in and out at runtime. This creates a highly dynamic system, further increasing flexibility and decreasing coupling. Many systems take advantage of this ability, such as Spring itself.

In summary, each layer is exposed as an interface. The interface provides a layer of abstraction, making it easy to change the implementation of the layer without affecting the rest of the application.

Layers in a Spring MVC Application

This section contains discussions of each major layer in a typical Spring MVC application. We will also cover some potential deviations from this design. A few discussions will touch on Spring MVC interfaces or classes. Do not fear; we explain each subject in detail in following chapters.

User Interface Layer

The user interface layer is responsible for presenting the application to the end user. This layer renders the response generated by the web layer into the form requested by the client. For instance, cell phones usually require WML or at least specializations of XHTML. Other clients may want PDFs for their user interface. And, of course, browsers want the response rendered as XHTML. We keep the user interface rendering layer separate from the web layer (discussed next) so that we can reuse the web layer as much as possible.

■ **Note** From the perspective of a web developer, the user interface layer is a very important level of abstraction. It's easy to consider the user interface as a sublayer below the full web layer, and this view is not incorrect. For the purposes of this book, specializing in web applications, we've elevated the user interface to a formal layer because it has its own set of concerns and implementation details.

The user interface layer is typically the top layer. Conceptually, this means it is the last layer in the processing chain before the bytes are sent to the client. By this point, all the business logic has been performed, all transactions are committed, and all resources have been released.

Being last, in this case, is a good thing. The user interface layer is responsible for rendering the bytes that are sent to the client. The client, in a web application, is remote and connected via an unreliable network.[3] The transfer of bytes can slow down, be repeated, or even stop. The UI layer is kept separate from the other layers because we want the system to continue to process other requests, with valuable resources such as database connections, without having to wait on network connections. In other words, the act of rendering a response for a client is separate from the act of gathering the response.

Another reason for isolating the user interface into its own layer is the practical reality that there are many toolkits for rendering the user interface. Some examples include JSP, Velocity, FreeMarker, and XSLT (all of which are well supported by Spring). Putting the UI concerns behind its own layer allows the rendering technology to change without affecting the other layers. The other layers of the system should be hidden from the choice of the rendering toolkit. There are simply too many options, each with its own pros and cons, to tie a particular toolkit directly into the system.

Teams with a dedicated UI specialist benefit greatly by separating this layer. UI designers typically work with a different toolset and are focused on a different set of concerns than the developers. Providing them with a layer dedicated to their needs shields them from the internal details of much of the system. This is especially important during the prototyping and interface design stages of development.

Spring MVC's User Interface Layer

Spring MVC does a nice job of isolating the UI concerns into a few key interfaces. The `org.springframework.web.servlet.View` interface represents a view, or page, of the web application. It is responsible for converting the result of the client requested operation (the response model) into a form viewable by the client.

■ **Note** The model is a collection of named objects. Any object that is to be rendered by the `View` is placed into the model. The model is purposely generic so that it may work with any view rendering technology. The view rendering toolkit is responsible for rendering each object in the model.

3. See fallacy number one, "The network is reliable," in *The Eight Fallacies of Distributed Computing* by Peter Deutsch (`http://today.java.net/jag/Fallacies.html`).

The `View` interface is completely generic and has no specific view rendering dependencies. Each view technology will provide an implementation of this interface. Spring MVC natively supports JSP, FreeMarker, Velocity, XSLT, JasperReports, Excel, and PDF.

The `org.springframework.web.servlet.ViewResolver` provides a helpful layer of indirection. The `ViewResolver` provides a map between view instances and their logical names. For instance, a JSP page with a filename of `/WEB-INF/jsp/onSuccess.jsp` can be referred to via the name "success". This decouples the actual `View` instance from the code referencing it. It's even possible to chain multiple `ViewResolvers` together to further create a flexible configuration.

Dependencies

The view layer typically has a dependency on the domain model (see the following discussion). This is not always the case, but often it is very convenient to directly expose and render the domain model. Much of the convenience of using Spring MVC for form processing comes from the fact that the view is working directly with a domain object.

For example, the model is typically filled with instances from the domain model. The view technology will then render the pages by directly querying the domain model instances.

Some may argue this creates an unnecessary coupling in the system. We believe that the alternatives create such an inconvenience that it outweighs any benefits of divorcing the two layers. For example, Struts promotes a class hierarchy for its model beans that is completely separate from the domain model. This creates an odd parallel class hierarchy, with much duplicated effort. Often the system isn't that decoupled because the view-specific classes are nearly one-to-one reflections on the domain classes.

To keep things simple, Spring MVC promotes integrating the domain classes to the view. We consider this acceptable, but in no way is it enforced or required.

Summary

The user interface layer (also known as the *view*) is responsible for rendering output for the client. Typically, this means XHTML for a web application, but Spring MVC supports many different view rendering technologies for both text and binary output. The key interfaces are `org.springframework.web.servlet.View` (representing a single page) and `org.springframework.web.servlet.ViewResolver` (providing a mapping between views and logical names). We cover Spring MVC's view technology in Chapter 7 and 8.

Web Layer

Navigation logic is one of two important functions handled by the web layer. It is responsible for driving the user through the correct page views in the correct order. This can be as simple as mapping a single URL to a single page or as complex as a full work flow engine.

Managing the user experience and travels through the site is a unique responsibility of the web layer. Many of the layers throughout this chapter assume much more of a stateless role. The web layer, however, typically does contain some state to help guide the user through the correct path.

There typically isn't any navigation logic in the domain model or service layer; it is the sole domain of the web layer. This creates a more flexible design, because the individual functions of the domain model can be combined in many different ways to create many different user experiences.

The web layer's second main function is to provide the glue between the service layer and the world of HTTP. It becomes a thin layer, delegating to the service layer for all coordination of the business logic. The web layer is concerned with request parameters, HTTP session handling, HTTP response codes, and generally interacting with the Servlet API.

The HTTP world is populated with request parameters, HTTP headers, and cookies. These aspects are not business logic–specific, and thus are kept isolated from the service layer. The web layer hides the details of the web world from the business logic.

Moving the web concerns out of the business logic makes the core logic very easy to test. You won't be worrying about setting request variables, session variables, HTTP response codes, or the like when testing the business layer. Likewise, when testing the web layer, you can easily mock the business layer and worry only about issues such as request parameters. Chapter 10 offers a more detailed discussion on testing the web layer.

Divorcing the web concerns from the service layer also means the system can export the same business logic via multiple methods. This reduces code duplication and allows the system to easily add connection mechanisms, such as HTTP, SOAP, or XML-RPC, quickly and easily. The web layer becomes just another client connection mechanism, providing access to core business functionality, but never implementing the functionality directly.

The web layer can be implemented as simply as servlets, for instance. These servlets will perform the work of turning request parameters into meaningful objects for the service layer, and then calling a method on a service interface. The web layer is also responsible, among other things, for turning any business exceptions into appropriate error messages for end users.

Higher-level frameworks, such as Spring MVC and Tapestry, offer sophisticated mechanisms for this translation between the raw request parameters and the business logic layer. For instance, Spring MVC will map request parameters onto plain old Java objects (POJOs) that the business logic can operate on directly. Spring MVC also implements sophisticated work flows for processing requests, structuring the way the request is handled and making extension easy.

There are two main types of web layer implementations: request/response frameworks and component frameworks. A request/response framework is built to interact directly with the Servlet API and the `HttpServletRequest` and the `HttpServletResponse`. These types of frameworks are considered to have a push model, because the user code will compile a result and then push it out to be rendered. Spring MVC is considered a request/response framework.

Other frameworks have adopted different approaches to processing a web request. Some frameworks, such as Tapestry and JavaServer Faces (JSF), are considered component-based. Those frameworks attempt to not only hide the Servlet API from you, but also make programming for the web feel like programming a Swing application. Those frameworks are essentially event-driven, as the components respond to events originally coming from the web layer.

Both types of programming models have their advantages and disadvantages. We believe Spring MVC is a good balance. It provides a rich hierarchy of implementations for handling requests, from the very basic to the very complex. You can choose how tightly you wish to couple yourself to the Servlet API. Using the base Spring MVC classes does expose you to the Servlet API. On the other hand, you will see that with Spring Web Flow or the `ThrowawayController`, the Servlet API can be hidden completely. As with many things in the Spring Framework, the developer is left to choose what is best for that particular situation.

Dependencies

The web layer is dependent on the service layer and the domain model. The web layer will delegate its processing to the service layer, and it is responsible for converting information sent in from the web to domain objects sufficient for calls into the service layer.

Spring MVC Web Layer

Spring MVC provides an `org.springframework.web.servlet.mvc.Controller` interface and a very rich class hierarchy below it for its web layer contract. Put very simply, the `Controller` is responsible for accepting the `HttpServletRequest` and the `HttpServletResponse`, performing some unit of work, and passing off control to a `View`. At first glance, the `Controller` looks a lot like a standard servlet. On closer inspection, the `Controller` interface has many rich implementations and a more complete life cycle.

Out of the box, Spring MVC provides many `Controller` implementations, each varying in its complexity. For instance, the `Controller` interface simply provides a method analogous to the servlet's `doService` method, assisting very little in the way of navigation logic. On the other hand, the `SimpleFormController` implements a full single-form work flow, from initial view of the form, to validation, to form submission. For very complex work flows and user experiences, Spring Web Flow provides a declarative means to navigate a user through a set of actions. Spring Web Flow contains a full-featured state machine so that all of the logic for a user's path through the system is moved out of the `Controller`s. This simplifies the wiring and configuration of complex work flows.

When a `Controller` wants to return information to the client, it populates a `ModelAndView`. The `ModelAndView` encapsulates two pieces of information. It contains the model for the response, which is merely a `Map` of all the data that makes up the response. It also contains a `View` reference, or the reference name for a `View` (to be looked up by a `ViewResolver`).

Summary

The web layer manages the user's navigation through the site. It also acts as the glue between the service layer and the details of the Servlet API.

Spring MVC provides a rich library of implementations of the `Controller` interface. For very complex user work flows, Spring Web Flow builds a powerful state machine to manage a user's navigation.

Service Layer

The service layer plays very important roles for the both the client and the system. For the client, it exposes and encapsulates coarse-grained system functionality (use cases) for easy client usage. A method is *coarse grained* when it is very high level, encapsulating a broad work flow and shielding the client from many small interactions with the system. The service layer should be the only way a client can interact with the system, keeping coupling low because the client is shielded from all the POJO interactions that implement the use case.

For the system, the service layer's methods represent transactional units of work. This means with one method call, many POJOs and their interactions will be performed under a single transaction. Performing all the work inside the service layer keeps communication between the client and the system to a minimum (in fact, down to one single call). In a highly

transactional system, this is important to keep transaction life span to a minimum. As an added benefit, moving the transactions to a single layer makes it easy to centralize the transaction configurations.

Each method in the service layer should be stateless. That is, each call to a service method creates no state on the object implementing the service interface. No single method call on a service object should assume any previous method calls to itself. Any state across method calls is kept in the domain model.

In a typical Spring MVC application, a single service layer object will handle many concurrent threads of execution, so remaining stateless is the only way to avoid one thread clobbering another. This actually leads to a much more simple design, because it eliminates the need to pool the service objects. This design performs much better than a pool of instances, because there is no management of checking the object in and out of the pool. Using a singleton for each service object keeps memory usage to a minimum as well.

This layer attempts to provide encapsulations of all the use cases of the system. A single use case is often one transactional unit of work, and so it makes sense these two aspects are found in the service layer. It also makes it easy to refer to one layer for all the high-level system functionality.

Consolidating the units of work behind a service layer creates a single point of entry into the system for end users and clients. It now becomes trivial to attach multiple client communication mechanisms to a single service interface. For instance, with Spring's remoting capabilities, you can expose the same service via SOAP, RMI, Java serialization over HTTP, and, of course, standard XHTML. This promotes code reuse and the all-important DRY (Don't Repeat Yourself) principle by decoupling the transactional unit of work from the transport or user interface. For more information on Spring's remoting capabilities, refer to *Pro Spring* by Rob Harrop (Apress, 2005) or to the online documentation (`http://www.springframework.org/documentation`).

Example

As just mentioned, the service layer provides an interface for clients. A typical interface has very coarse-grained methods and usually looks something like Listing 3-1.

Listing 3-1. *Coarse-Grained Service Layer Interface*

```
public interface AccountManager {
  void activateAccount(String accountId);

  void deactivateAccount(String accountId);

  Account findAccountByUsername(String username);
}
```

You can see why these methods are considered coarse grained. It takes one simple call for the client to achieve completion of a single use case. Contrast this to a fine-grained interface (see Listing 3-2), where it would take many calls, to potentially many different objects, to accomplish a use case.

Listing 3-2. *Fine-Grained Service Layer Interface*

```
public interface FineGrainedAccountManager {
  Account findAccountByUsername(String username);

  void setAccountToActive(Account account);

  boolean accountAbleToBeActivated(Account account);

  void sendActivationEmail(Account account, Mailer mailer);
}
```

With the preceding example, too much responsibility is given to the client. What is the correct order of the method calls? What happens if there is a failure? There's no way to guarantee that the same account instance is used for every call. This fine-grained interface couples the client too closely to *how* the use case is implemented.

In environments where the client is remote, a coarse-grained interface is an important design element. Serialization is not a cheap operation, so it is important to serialize at most once per call into the service layer. Even in systems where the client is in the same virtual machine, this layer plays a crucial role when separating the concerns of the system and making it easier to decouple and test.

Dependencies

The service layer is dependent upon the domain model and the persistence layer, which we discuss in the following sections. It combines and coordinates calls to both the data access objects and the domain model objects. The service layer should never have a dependency on the view or web layers.

It is important to note that it is usually unnecessary for this layer to have any dependencies on framework-specific code, or infrastructure code such as transaction management. The Spring Framework does a good job of transparently introducing system aspects so that your code remains highly decoupled.

Spring's Support for the Service Layer

The Spring Framework does provide any interfaces or classes for implementing the business aspects of the service layer. This should not be surprising, because the service layer is specific to the application.

Instead of defining your business interfaces, Spring will help with the programming model. Typically, the Spring Framework's `ApplicationContext` will inject instances of the service into the web `Controllers`. Spring will also enhance your service layer with services such as transaction management, performance monitoring, and even pooling if you decide you need it.

Summary

The service layer provides a stateless, coarse-grained interface for clients to use for system interaction. Each method in the service layer typically represents one use case. Each method is also one transactional unit of work.

Using a service layer also keeps coupling low between the system and the client. It reduces the amount of calls required for a use case, making the system simpler to use. In a remote environment, this dramatically improves performance.

Domain Model Layer

The domain object model is the most important layer in the system. This layer contains the business logic of the system, and thus, the true implementation of the use cases. The domain model is the collection of nouns in the system, implemented as POJOs. These nouns, such as `User`, `Address`, and `ShoppingCart`, contain both state (user's first name, user's last name) and behavior (`shoppingCart.purchase()`). Centralizing the business logic inside POJOs makes it possible to take advantage of core object-oriented principles and practices, such as polymorphism and inheritance.

■ **Note** We've talked a lot about interfaces and how they provide good contracts for layer interaction. Interfaces are used tremendously with the service layer but aren't as common inside the domain model. The use of interfaces inside the domain model should be driven by pure object-oriented design considerations. Add them into the domain model when it makes sense, but don't feel obligated to add interfaces merely to put interfaces in front of everything.

When we say *business logic*, what do we mean? Any logic the system performs to satisfy some rule or constraint dictated by the customer is considered business logic. This can include anything from complex state verification to simple validation rules. Even a seemingly simple CRUD (create, read, update, and delete) application will have some level of business logic in the form of database constraints.

You might have noticed a contradiction just now. Can you spot it? Earlier, we advocated that the domain model should encapsulate the business logic of the system. Yet we have acknowledged that there are some business rules that live in the database, in the form of constraints such as UNIQUE or NOT NULL. While you should strive to put all your business logic inside your domain model, there are cases where the logic will live in other places. We consider this split to be acceptable, because the database does a good job at enforcing these constraints. You can, however, continue to express the business rule found in the database in your domain model. This way, the rule won't be hidden.

For example, let's say that all emails must be unique in the system. We could code this logic into the domain model by loading up all the users and the searching through each one. The performance on this type of operation, however, would be horrible. The database can handle this sort of data integrity requirement with ease. The moral of the story is that the domain model should contain most of the business logic, but place the logic outside the model when there is a good reason.

It's important to note that business logic does not mean just a strong set of relationships between objects. A domain model that contains only state and relationships to other models is what Martin Fowler would call an Anemic Domain Model (`http://www.martinfowler.com/bliki/AnemicDomainModel.html`). This anti-pattern is found when there is a rich domain model that doesn't seem to perform any work. It might be tempting to place all your business logic

into the service layer or even the web layer, but this will negate any benefits of an object-oriented system. Remember, objects have state and behavior.

Dependencies

The domain model, or business object model, is a good example of a layer that largely permeates the other layers. It may be helpful to think of the domain model as a vertical layer. In other words, many other layers have dependencies on the domain model. It is important to note, however, that the object has no dependencies on any other layer.

The domain model should never have any dependencies on the framework, so that it can decouple itself from the environment it will be hosted in. First and foremost, this means the business logic can be tested outside of the container and independently of the framework. This speeds up development tremendously, as no deployments are required for testing. Unit tests become very simple to create, as they are testing simple Java code, without any reliance on database connections, web frameworks, or the other layers in the system.

All of the other layers have a dependency on the domain model. For instance, the service layer typically combines multiple methods from the domain model together to run under one transaction. The user interface layer might serialize the domain model for a client into XML or XHTML. The data access layer is responsible for persisting and retrieving instances of the objects from the model.

As you can see, each layer is responsible for their problem domains, but they all live to service the domain model. The domain model is the first-class citizen of the system, and frameworks like Spring and Spring MVC support this notion. Developing web applications with Spring MVC is refreshing, because there is so much true object-oriented development.

Spring's Support for the Domain Layer

Just like the service layer, Spring does not provide any base interfaces for your object model. Doing so would be completely against the ideologies of a lightweight container such as Spring. However, Spring does provide certain convenience interfaces one might choose to use when needing to integrate tightly with the framework. The need to do so is rare, and we caution against introducing any framework-specific interfaces into your base object model.

Spring can also enhance your domain model via AOP, just like it will with the service layer. To Spring, both the service layer and domain model are simply a set of POJOs.

If you decide to, Spring will perform Dependency Injection on your domain model as well. This is an advanced technique, but it is recommended for advanced object-oriented domain models.

■ **Tip** Two ways of Dependency Injecting your domain model objects include an AspectJ approach (`http://www.aspectprogrammer.org/blogs/adrian/2005/03/hacking_with_ha.html`) and a Hibernate Interceptor approach (`org.springframework.orm.hibernate.support.DependencyInjectionInterceptorFactoryBean`, currently in the sandbox).

Dependency Injection for the Domain Model

As you probably know, Spring does an excellent job of creating POJOs and wiring them together. This works well when the object is actually created and initialized by Spring, but this isn't always the case with objects in the domain model. These instances can come from outside the `ApplicationContext`, for instance loaded directly by the database. How do we inject these objects with their dependencies before they enter the application?

If you have an object instance already constructed, but you would like Spring to wire that object with any dependencies, you will need an instance of a `AutowireCapableBeanFactory`. Luckily, `XmlBeanFactory` happens to implement that interface. Listing 3-3 illustrates how to wire an existing POJO with dependencies.

Listing 3-3. *Bean Definition for Wiring an Existing POJO*

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

  <bean id="account" abstract="true"
    class="com.apress.expertspringmvc.chap3.Account">
      <property name="mailSender" ref="mailSender" />
  </bean>

  <bean id="mailSender"
    class="org.springframework.mail.javamail.JavaMailSenderImpl">
      <property name="host" value="mail.example.com" />
  </bean>

</beans>
```

The preceding bean definition specifies one abstract bean definition for an `Account` object. An `Account` has a `MailSender` as a property. We then define the `MailSender` as the second bean in this bean definition. Note we use `abstract="true"` to ensure it will never be created inside the container.

Listing 3-4. *Simple POJO Requiring an External Resource*

```
package com.apress.expertspringmvc.chap3;

import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

public class Account {

    private String email;

    private MailSender mailSender;
```

```
    private boolean active = false;

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void activate() {
        if (active) {
            throw new IllegalStateException("Already active");
        }
        active = true;
        sendActivationEmail();
    }

    private void sendActivationEmail() {
        SimpleMailMessage msg = new SimpleMailMessage();
        msg.setTo(email);
        msg.setSubject("Congrats!");
        msg.setText("You're the best.");
        mailSender.send(msg);
    }
}
```

The `Account` POJO in Listing 3-4 has a method called `activate()` that sets the `account` instance as active and then sends an activation email. Clearly it needs an instance of `MailSender`, as it doesn't create one itself. We will use the code in Listing 3-5 to ask Spring to inject this dependency, based on the previous abstract `account` definition.

Listing 3-5. *Example Dependency Injection of Existing POJO*

```
package com.apress.expertspringmvc.chap3;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.AutowireCapableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class DependencyInjectionExistingPojo {

    public static void main(String[] args) throws Exception {
```

```
        BeanFactory beanFactory = new XmlBeanFactory(
                new ClassPathResource("chapter3.xml"));

        Account account = new Account();
        account.setEmail("email@example.com");

        ((AutowireCapableBeanFactory)beanFactory).applyBeanPropertyValues(
                account, "accountPrototype");

        account.activate();
    }

}
```

The code in Listing 3-5 uses the `applyBeanPropertyValues()` method on `AutowireCapableBeanFactory` to apply a bean definition's properties to an existing bean. We are linking the `accountPrototype` bean definition from the XML file to the account instance. The `activate()` method now will work, because the `Account` object has its `MailSender` dependency.

The `AutowireCapableBeanFactory` interface also defines an `autowireBeanProperties()` method if you don't want to specify an abstract bean definition. This method will use an autowire strategy of your choice to satisfy any dependencies of the object.

Although this is a good example, most applications aren't quite this simple. The biggest issue will be the account instance, as it will most likely come from the database. Depending on what persistence mechanism you choose, you will want to see if it's possible to intercept the object after it is loaded from the database, but before it is sent back into the application. You can then apply this technique to inject the object with any extra dependencies.

■ **Tip** If you are using Hibernate, there is a `DependencyInjectionInterceptorFactoryBean` in Spring's sandbox that will wire objects loaded from Hibernate automatically. It provides a very nice way to transparently inject dependencies into your Hibernate objects as they come out of persistence. You can find this class in the `org.springframework.orm.hibernate.support` package.

Using this technique, you can build a very strong domain model that can support complex business logic. Be careful what you inject into your domain model; you don't want to increase the amount of dependencies the domain model has. For example, the domain model shouldn't know anything about the persistence layer. Let the service layer handle that coordination. You should only be injecting objects that help to implement business logic and rules.

Data Access Layer

The data access layer is responsible for interfacing with the persistence mechanism to store and retrieve instances of the object model. The typical CRUD methods are implemented by this layer.

The data access functionality gets its own layer for two reasons. Delegating the persistence functionality into its own layer protects the system from change and keeps tests quick to run and easy to write.

One of the primary reasons for abstraction in object-oriented systems is to isolate sections of the applications from change. The data access functionality is no different, and it is designed to isolate the system from changes in the persistence mechanisms.

As an example, a business requirement change might force all user accounts to be stored inside an LDAP-compliant directory instead of a relational database. While this might happen rarely, abstracting the persistence operations behind a single interface makes this a low-impact change for the system.

A more likely scenario is a change in the data access layer's implementation and libraries. Many different types of implementations are available, ranging from straight Java Database Connectivity (JDBC) to full-fledged object relational mapping frameworks such as Hibernate. Each offers its own distinct advantages, but they function in significantly different ways. When all data access is delegated to its own layer, changing from one persistence mechanism to another becomes possible. Again, it's unlikely that the persistence framework will be swapped out in a production system, but it's certainly possible.

Building the system to cope with change is important, for they say that we are building tomorrow's legacy software today. Recognizing a discrete problem domain of the system, such as data access, is important. Isolating those problem domains in their own interfaces and layers helps to keep the system adaptable in the face of change.

Keeping the time to run the system tests low is the other key reason the data access layer is isolated. Any unit test that requires more than the method being tested ceases to be a unit test and becomes an integration test. Unit tests are meant to test very small units of code. If the tests had to rely on an external database, then the code under test would not be isolated. If a problem would arise, both the external database and the actual code would have to be checked for problems.

Database connections are expensive resources to create and maintain. Unit tests should be very quick to run, and they will slow down tremendously if they require connections to the RDBMS. Isolating all persistence operations to one layer makes it easy to mock those operations, keeping test runs fast.

The system is built on a solid object model, and therefore the bulk of the unit tests will be against the object model. The data access features are in their own layer to keep the object model from having to manage a concern that is primarily orthogonal to the concern of implementing business logic.

Dependencies

It is important to note that, typically, only the service layer has a dependency on the data access layer. This means that there is typically only one layer that knows anything about the data access layer. There are two reasons for this.

The service layer implements transactional boundaries. The data access layer is concerned only with interacting with the persistence mechanism, and the persistence mechanism is typically transactional. Performing the data access operations within the scope of the service façade means that the current transaction is easily propagated to the data access layer.

Also, the service layer encapsulates many little operations from the POJOs, thus shielding the client from the inner workings of the system. Those POJOs have to be loaded from persistence. From a practical standpoint, the service layer coordinates the data access layer and the POJO layer such that the appropriate POJOs are loaded and persisted for the use case.

It's important to note that the persistence layer does not have to be accessed behind the service layer. The persistence code is just another set of JavaBeans. For very simple applications, directly accessing the Data Access Objects (DAOs) is not necessarily a bad thing. Be aware of handling transactional boundaries correctly, and be sure your domain model does not have any references to the DAOs.

Spring Framework Data Access Layer

The Spring Framework really shines when providing data access interfaces. The framework doesn't have a single common interface for all data access operations, because each toolkit (Hibernate, JDBC, iBATIS, and so on) is so varied. Your business needs will usually dictate the interface design. However, Spring does provide common patterns for interacting with the data access layer. For example, the template pattern is often used by data access operations to shield the implementer from common initialization and cleanup code. You will find template implementations for Hibernate (`HibernateTemplate`), JDBC (`JdbcTemplate`), iBATIS (`SqlMapTemplate`), and others inside the `org.springframework.jdbc` and `org.springframework.orm` packages.

One of the main benefits of Spring is its very rich and deep data access exception hierarchy. The framework can convert all of your database server's specific exceptions into a semantically rich exception, common across all database implementations. For instance, your database server's cryptic "Error 223—Foreign Key Not Found" will be converted to a strongly typed `DataIntegrityViolationException`. All of the exceptions in the `DataAccessException` hierarchy are of type `RuntimeException` to help keep code clean. These exceptions are commonly mapped across both database servers as well as persistence mechanisms. That is, `HibernateTemplate`, `JdbcTemplate`, and the others will throw the same set of exceptions. This helps tremendously with any potential porting required in the future.

For a full discussion of Spring's support for persistence, refer to Rob Harrop's *Pro Spring* (Apress, 2005), or the online documentation. This is a very valuable and powerful aspect of the framework.

For your application-specific code, typically you will first design a data access layer interface independently of any implementation or even the framework. For example, one such DAO interface might look like this:

```
public interface AccountDao {
    public Account findById(String accountId);
    public void deleteAccount(Account account);
    public void saveAccount(Account account);
}
```

The operations in this interface do not mention any persistence technology. This keeps coupling low, because clients of the data access layer will be interacting through this interface. The clients don't care how the data is persisted or retrieved, and the interface shields them from any underlying technologies.

Once you have defined your DAO interface, it is easy to choose one of Spring's convenience classes, such as `HibernateDaoSupport`, to subclass and implement. This way, your class takes advantage of Spring's data access support, while implementing your system's specific DAO contract.

Options: There's More Than One Way to Do It

Is this the only way to construct a Spring MVC application? Are all those layers needed all the time? It is important to note that the preceding discussions are suggestions and guidelines. Many successful web applications don't follow this same pattern.

When choosing the architecture of the system, it's important to recognize what type of application is being built. Will the application live for a long time? Is the application for internal or external use? How many developers will be maintaining the application? Understanding what the initial investment will be will help to dictate the application architecture. You must correctly balance the needs of today with the inevitable needs of tomorrow's growth and maintenance.

We understand that if the web application starts its life as a single page with a single SQL query, implementing all the layers would be overkill. We also understand that applications have a tendency to grow in scope and size. The important steps during development are refactoring constantly and writing unit tests consistently. Although you may not start out with an *n*-layer application, you should refactor toward that goal.

Spring MVC applications certainly encourage your applications to head a certain direction, but they by no means require it. Letting the developer choose what is best for the application is what the Spring Framework and Spring MVC is all about.

No matter how your web application is implemented, it's important to take a few points to heart. Consider using layers in your application to help isolate separate areas of concern. For instance, do not put JDBC code directly in your servlets. This will increase coupling in your application. Separating into layers will structure your application, making it easier to learn and read.

Use interfaces as a means to hide clients from implementations at layer boundaries. This also reduces coupling and increases testability. You can accomplish this very simply these days, because modern IDEs support refactoring techniques such as Extract Interface. Interfaces are especially helpful when used at integration points.

Most importantly, you should put business logic inside POJOs. This exposes the full power of OOP for your domain model. For instance, encapsulate all the business rules and logic to activate an account inside an `activate()` method on the `Account` class (as we did in the "Domain Model Layer" section of this chapter). Think about how to apply common OO features such as polymorphism and inheritance to help solve the business problems. Focus first on the domain model, and accurately reflect the problem domain by building classes with state and behavior. Also, don't let system-wide, non–business-specific concerns like transaction management or logging creep into the domain model. Let the Spring Framework introduce those aspects via AOP.

These principles are not new, and they have been preached for a long time in the object-oriented world. Until the arrival of Inversion of Control containers, with strong Dependency Injection support, these principles were very difficult to implement in medium to large systems. If you have built systems before and had to throw out many of your hard-learned OOP practices, you will soon breathe a welcome sigh of relief. The Spring Framework makes it possible to integrate and develop loosely coupled web application systems.

Summary

We've shown that a typical Spring MVC application has many layers. You will write code to handle the user interface, the web navigation, the service layer, the domain model, and the persistence layer. Each layer is isolated in such a way to reduce coupling and increase testability. The layers use interfaces as their contracts, shielding other layers from implementation details. This allows the layers to change independent of the rest of the system. The system becomes more testable, as each layer can be tested in isolation. The other layers are mocked in the unit tests, keeping test runs quick and focused on testing only the target code.

We've also shown that the most important layer is the object model. The object model contains the business logic of the system. All the other layers play supporting roles and handle orthogonal system concerns such as persistence or transactions. The web layer is kept thin, implementing no business logic and providing a bridge between the world of the web and the object model.

A main goal has been to keep the framework out of our code as much as possible. By using the Spring Framework, we are able to keep framework-specific code out of our object model completely. The interfaces to our data access layer are unaware of any framework. The service façade layer's interfaces are also devoid of any framework code. The Spring Framework binds our layers together transparently, without dictating application design or implementation.