CHAPTER 5

The Processing Pipeline

Spring MVC applications are highly configurable and extensible, and most of that power comes from its software architecture. Inside Spring MVC, as is the case with the Spring Framework as a whole, interfaces and abstractions are provided so that you can easily customize the behavior and work flow of your web application. In this chapter we will examine the `DispatcherServlet` and the processing pipeline that it controls in order to understand how a request is handled, as well as how best to tap into the many extension points and life cycle events.

Processing Requests

A new HTTP request entering the system is passed to many different handlers, each playing its own small part in the overall processing of the request. We will now look at the timeline of a new request and the order in which the handlers are activated.

Request Work Flow

1. Discover the request's `Locale`; expose for later usage.
2. If the request is a multipart request (for file uploads), the file upload data is exposed for later processing.
3. Locate which request handler is responsible for this request (e.g., a `Controller`).
4. Locate any request interceptors for this request. Interceptors are like filters, but customized for Spring MVC.
5. Call `preHandle()` methods on any interceptors, which may circumvent the normal processing order (see "HandlerInterceptors" in Chapter 6).
6. Invoke the `Controller`.
7. Call `postHandle()` methods on any interceptors.
8. If there is any exception, handle it with a `HandlerExceptionResolver`.
9. If no exceptions were thrown, and the `Controller` returned a `ModelAndView`, then render the view. When rendering the view, first resolve the view name to a `View` instance.
10. Call `afterCompletion()` methods on any interceptors.

Functionality Overview

As you can see, the `DispatcherServlet` provides Spring MVC much more functionality than `Controllers` and `Views`. The main theme here is pluggability, as each piece of functionality is abstracted behind a convenient interface. We will visit each of these areas with more depth later in this chapter, but for now let's look at what Spring MVC is really capable of.

Locale Aware

Especially important with applications sensitive to internationalization (i18n) issues, Spring MVC binds a `Locale` to all requests. Typically, the servlet container will set the `Locale` by looking at HTTP headers sent by the client, but Spring MVC abstracts this process and allows for the `Locale` to be retrieved and stored in arbitrary ways. By extending the `LocaleResolver` interface, you can discover and set the `Locale` for each request based on your application's requirements. The `Locale` is then available during the entire request processing, including view rendering.

■ **Tip** See section 14.4 of the HTTP RFC for more on the Accept-Language header: `http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.4`.

Multipart File Uploads

A standard functionality requirement for all web application frameworks, file uploads (also known as multipart requests) are handled in a pluggable manner. Spring MVC integrates with the two well-known Java file upload libraries, Jason Hunter's COS library (`http://www.servlets.com/cos`, from his book *Java Servlet Programming* (O'Reilly, 2001)) and Jakarta Commons' FileUpload library (`http://jakarta.apache.org/commons/fileupload`). If neither of these libraries covers your application's needs, you may extend the `MultipartResolver` interface to implement your custom file upload logic.

Request HandlerAdapters

While all the examples in this book will cover `Controllers` as the primary way to handle incoming requests, Spring MVC provides an extension point to integrate any request handling device. The `HandlerAdapter` interface, an implementation of the adapter pattern, is provided for third-party HTTP request handling framework integration.

■ **Tip** Learn more about the adapter pattern, which adapts one system's API to be compatible with another's, inside the book *Design Patterns: Elements of Reusable Object-Oriented Design* (Gamma, Helm, Johnson, and Vlissides; Addison Wesley, 1995).

Mapping Requests to Controllers

The `HandlerMapping` interface provides the abstraction for mapping requests to their handlers. Spring MVC includes many implementations and can chain them together to create very flexible and partitioned mapping configurations. Typically a request is mapped to a handler (`Controller`) via a URL, but other implementations could use cookies, request parameters, or external factors such as time of day.

Intercepting Requests

Like servlet filters wrapping one or more servlets, `HandlerInterceptors` wrap request handlers and provide explicit ways to execute common code across many handlers. `HandlerInterceptors` provide useful life cycle methods, much more fine grained than a filter's simple `doFilter()` method. An interceptor can run before a request handler runs, after a request handler finishes, and after the view is rendered. Like servlet filters, you may wrap a single request handler with multiple interceptors.

Custom Exception Handling

Spring MVC allows for more exact exception handling than the standard `web.xml` file through its `HandlerExceptionResolver` interface. It's still possible to simply map exceptions to error pages, but with a `HandlerExceptionResolver` your exception mappings can be specific to the request handler plus the exception thrown. It's possible to chain these resolvers to create very specific exception handling mappings.

View Mapping

The extremely flexible view mapping mechanism, through the `ViewResolver` interface, is one of Spring MVC's most useful features. `ViewResolvers` are `Locale` aware, converting a logical view name into a physical `View` instance. Complex web applications are not limited to a single view technology; therefore Spring MVC allows for multiple, concurrent view rendering toolkits.

Pieces of the Puzzle

As you can see, the request is passed between quite a few different processing elements. While this might look confusing, Spring MVC does a good job hiding this work flow from your code. The work flow (see "Request Work Flow" earlier in this chapter) is encapsulated inside the `DispatcherServlet`, which delegates to many different components providing for easy extension and customization.

DispatcherServlet

As mentioned in Chapter 4, the `DispatcherServlet` is the front controller of the web application. It gets its name from the fact that it dispatches the request to many different components, each an abstraction of the processing pipeline.

Declaration

Typically, you will only declare and configure this class. All the customization is done through configuring different delegates instead of extending or modifying this class directly.

Caution The `DispatcherServlet` will be marked `final` in the near future, so avoid subclassing this class.

You saw the declaration and configuration of this servlet in Chapter 4. To quickly review, this servlet is configured in your application's `web.xml` file, as shown in Listing 5-1.

Listing 5-1. *DispatcherServlet in the web.xml*

```
<servlet>
  <servlet-name>spring</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>spring</servlet-name>
  <url-pattern>/app/*</url-pattern>
</servlet-mapping>
```

Of course, the URL pattern you choose for the `servlet-mapping` element is up to you.

■**Tip** Many servlet containers will validate the `web.xml` against its DTD or schema file, so be sure to place the elements in the right order and in the right place.

Initialization

When the `DispatcherServlet` initializes, it will search the `WebApplicationContext` for one or more instances of the elements that make up the processing pipeline (such as `ViewResolvers` or `HandlerMappings`).

■**Tip** Remember that the `WebApplicationContext` is a special `ApplicationContext` implementation that is aware of the servlet environment and the `ServletConfig` object.

For some of the component types such as `ViewResolvers` (see Table 5-1), the `DispatcherServlet` can be configured to locate all instances of the same type. The servlet will then chain the components together and order them, giving each the chance to handle the request.

■**Note** The `DispatcherServlet` uses the `Ordered` interface to sort many of its collections of delegates. To order anything that implements the `Ordered` interface, simply give it a property named `order`. The lower the number, the higher it will rank.

Usually, the first element to respond with a non-null value wins. This is very useful if your application requires different ways to resolve view names, for instance. This technique also allows you to create modular configurations of request handlers and then chain them together at runtime.

The `DispatcherServlet` searches for its components using the algorithm pictured in Figure 5-1. The path through the algorithm is dependent on many factors, including if multiple components of the same type can be detected and if there is a default strategy available if none are found in the `ApplicationContext`. For many types of components, if you disable the automatic detection by type, then the `DispatcherServlet` will fall back to searching for a single component with a well-known bean name.

Table 5-1 lists the discovery rules and interfaces of the components managed by the `DispatcherServlet`.

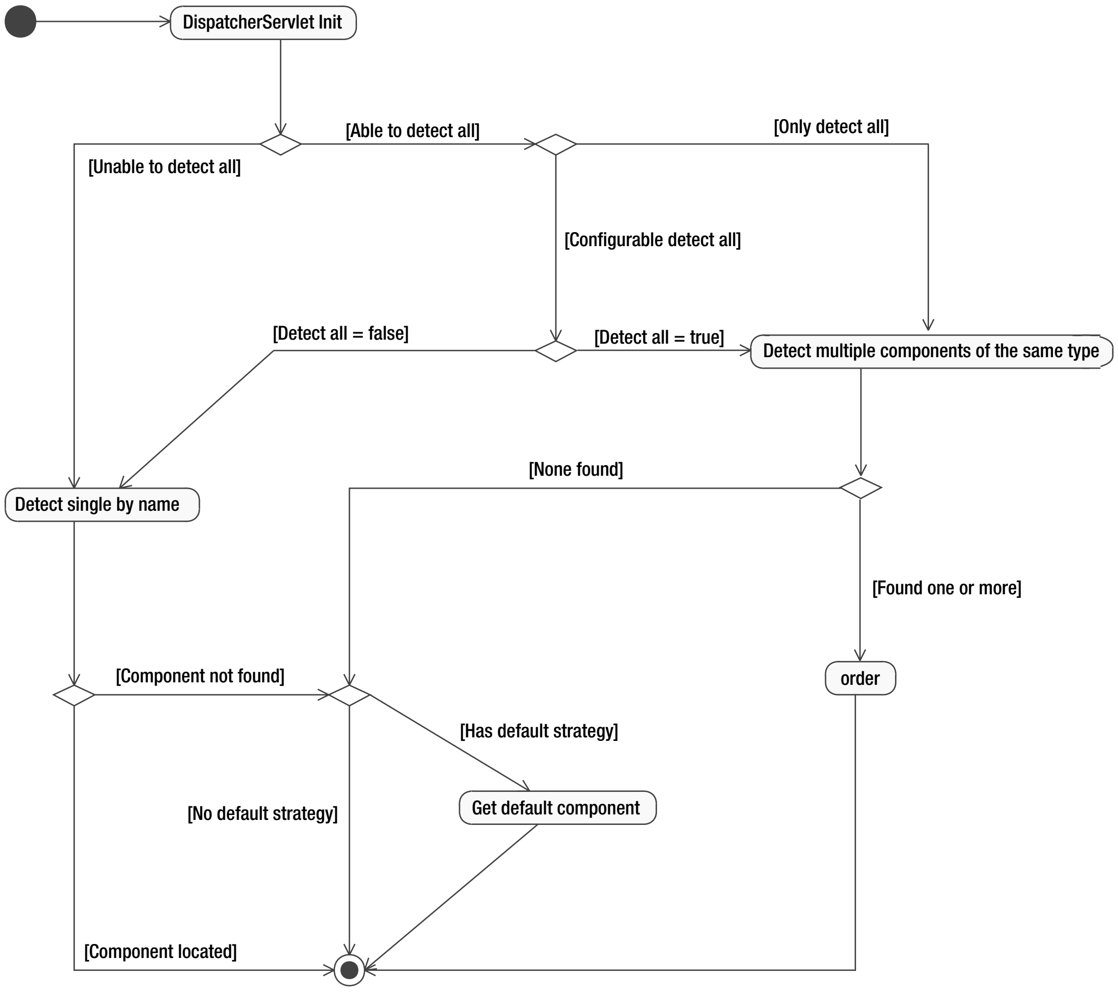

Figure 5-1. *DispatcherServlet discovery of components algorithm*

Table 5-1. *Interfaces of the Processing Pipeline*

Interface Name	Default Bean Name	Purpose	Chain Multiple?	Can Detect All?
`org.springframework.web.servlet.HandlerMapping`	`handlerMapping`	Maps requests to handlers (e.g., `Controllers`)	Yes	Yes
`org.springframework.web.servlet.HandlerAdapter`	none	Instance of adapter pattern, decouples handlers from the `DispatcherServlet`	Yes	No
`org.springframework.web.servlet.ViewResolver`	`viewResolver`	Maps view names to view instances	Yes	Yes
`org.springframework.web.servlet.HandlerExceptionResolver`	`handlerExceptionResolver`	Map exceptions to handlers and views	Yes	Yes
`org.springframework.web.multipart.MultipartResolver`	`multipartResolver`	Strategy interface to handle file uploads	No	No
`org.springframework.web.servlet.LocaleResolver`	`localeResolver`	Strategy interface for resolving the `Locale` of a request	No	No
`org.springframework.web.servlet.ThemeResolver`	`themeResolver`	Strategy interface for resolving a theme for this request (not used directly in the `DispatcherServlet`)	No	No

During initialization, the `DispatcherServlet` will look for all implementations by type of `HandlerAdapters`, `HandlerMappings`, `HandlerExceptionResolvers`, and `ViewResolvers`. However, you may turn off this behavior for all types but `HandlerAdapter` by setting to `false` the `detectAllHandlerMappings`, `detectAllHandlerExceptionResolvers`, or `detectAllViewResolvers` properties. To set one or more of these properties, you must use the `web.xml` where you initially declared the `DispatcherServlet`. Listing 5-2 shows an example of disabling the detection of all `ViewResolvers`.

■ **Note** At the time this was written, there is no way to turn off automatic detection of all `HandlerAdapter`s.

Listing 5-2. *Disable Detection of all View Resolvers*

```
<servlet>
  <servlet-name>spring</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>detectAllViewResolvers</param-name>
    <param-value>false</param-value>
  </init-param>
</servlet>
```

If you do disable the automatic discovery, you will then need to name at least one bean of each type with the default bean name. Consult Table 5-1 for each type's default bean name.

The `DispatcherServlet` is configured with default implementations for most of these interfaces. This means that if no implementations are found in the `ApplicationContext` (either by name or by type), the `DispatcherServlet` will create and use the following implementations:

■ **Caution** There is no default implementation for `MultipartResolver`, `HandlerExceptionResolver`, or `ViewResolver`.

- `org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping`
- `org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter`
- `org.springframework.web.servlet.view.InternalResourceViewResolver`
- `org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver`
- `org.springframework.web.servlet.theme.FixedThemeResolver`

Let's now take a closer look at each element in the processing pipeline, starting with the `HandlerAdapter` interface.

HandlerAdapter

The `org.springframework.web.servlet.HandlerAdapter` is a system level interface, allowing for low coupling between different request handlers and the `DispatcherServlet`. Using this interface, the `DispatcherServlet` can interact with any type of request handler, as long as a `HandlerAdapter` is configured.

Tip If your application consists of only `Controller`s, then you may safely ignore this section. `Controller`s are supported by default, and no explicit configurations for `HandlerAdapter`s are required. However, read on if you are interested in integrating a third-party framework into Spring MVC.

Why not just require all request handlers to implement some well-known interface? The `DispatcherServlet` is intended to work with any type of request handler, including third-party frameworks. Integrating disparate software packages is often difficult because the source code isn't available or is very difficult to change. The Adapter design pattern attempts to solve this problem by adapting the third party's interface to the client's expected interface. The seminal book *Design Patterns* (Gamma, Helm, Johnson, and Vlissides; Addison Wesley, 1995) defines this pattern's goal as follows: "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

Spring's `HandlerAdapter` achieves this adaptation by delegation. Listing 5-3 shows the `HandlerAdapter` interface.

Listing 5-3. *HandlerAdapter Interface*

```
package org.springframework.web.servlet;

public interface HandlerAdapter {

  boolean supports(Object handler);

  ModelAndView handle(HttpServletRequest request, HttpServletResponse response,
      Object handler) throws Exception;

  long getLastModified(HttpServletRequest request, Object handler);

}
```

The `DispatcherServlet` will check whether the `HandlerAdapter` supports a handler type with a call to `supports()`. If so, the `DispatcherServlet` will then ask the adapter to delegate the request to the handler via the `handle()` method. Notice how this interface is provided the handler instead of looking one up via the `ApplicationContext`.

If your application uses only `Controllers`, which nearly all Spring MVC applications do, then you will never see this interface. It is intended, along with its default subclass `SimpleControllerHandlerAdapter`, to be used by the framework internally. However, if you are intending to integrate an exotic web framework, you may use this class to integrate it into the DispatcherServlet.

Listing 5-4 provides a simple example of an implementation of a HandlerAdapter for some exotic web framework.

Listing 5-4. *Example HandlerAdapter*

```
public class ExoticFrameworkHandlerAdapter implements HandlerAdapter {

    public boolean supports(Object handler) {
        return (handler != null) && (handler instanceof ExoticFramework);
    }

    public ModelAndView handle(HttpServletRequest req, HttpServletResponse res,
        Object handler) throws Exception {
        ExoticResult result = ((ExoticFramework)handler).executeRequest(req, res);
        return adaptResult(result);
    }

    private ModelAndView adaptResult(ExoticResult result) {
        ModelAndView mav = new ModelAndView();
        mav.getModel().putAll(result.getObjectsToRender());
        return mav;
    }

    public long getLastModified(HttpServletRequest req, Object handler) {
        return -1;  // exotic framework doesn't support this
    }

}
```

Configuring the DispatcherServlet to use this HandlerAdapter is quite easy, as the DispatcherServlet by default looks into the ApplicationContext for all HandlerAdapters. It will find all adapters by their type, and it will order them, paying special attention to any adapters that implement the org.springframework.core.Ordered interface.

Listing 5-5 contains the bean definition for the ExoticFrameworkHandlerAdapter.

Listing 5-5. *ApplicationContext with ExoticFrameworkHandlerAdapter*

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC
  "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="exoticHandlerAdapter"
        class="com.apress.expertspringmvc.chap4.ExoticFrameworkHandlerAdapter"
    />
</beans>
```

It does not matter what the name of the HandlerAdapter is, because the DispatcherServlet will look for beans of type HandlerAdapter.

Note that by specifying any HandlerAdapter, the default SimpleControllerHandlerAdapter will not be used. If your application requires two or more HandlerAdapters, you will need to explicitly specify all HandlerAdapters, including the default.

Summary

The HandlerAdapter, an example of the Adapter design pattern, is a system-level interface to promote easy integration between the DispatcherServlet and third-party frameworks. Unless using third-party frameworks, this interface and its implementations are normally hidden from the developer. The DispatcherServlet will also chain multiple adapters if found in the ApplicationContext and will order them based on the Ordered interface.

HandlerMapping

No web application is complete without mapping its request handlers to URLs. As with all things in Spring MVC, there is no one way to map a URL to a Controller. In fact, it's very possible to create a mapping scheme and implementation that doesn't even rely on URLs at all. However, because the provided implementations are all based on URL paths, we will now review the default path matching rules.

Path Matching

Path matching in Spring MVC is much more flexible than a standard web.xml's servlet mappings. The default strategy for path matching is implemented by org.springframework.util.AntPathMatcher. As its name hints, path patterns are written using Apache Ant (http://ant.apache.org) style paths. Ant style paths have three types of wildcards (listed in Table 5-2), which can be combined to create many varied and flexible path patterns. See Table 5-3 for pattern examples.

Table 5-2. *Ant Wildcard Characters*

Wildcard	Description
?	Matches a single character
*	Matches zero or more characters
**	Matches zero or more directories

Table 5-3. *Example Ant-Style Path Patterns*

Path	Description
/app/*.x	Matches all .x files in the app directory
/app/p?ttern	Matches /app/pattern and /app/pXttern, but not /app/pttern
/**/example	Matches /app/example, /app/foo/example, and /example
/app/**/dir/file.*	Matches /app/dir/file.jsp, /app/foo/dir/file.html, /app/foo/bar/dir/file.pdf, and /app/dir/file.java
/**/*.jsp	Matches any .jsp file

Path Precedence

The ordering and precedence of the path patterns is not specified by any interface. However, the default implementation, found in `org.springframework.web.servlet.handler.AbstractUrlHandlerMapping`, will match a path based on the longest (most specific) matching pattern.

For example, given a request URL of `/app/dir/file.jsp` and two path patterns of `/**/*.jsp` and `/app/dir/*.jsp`, which path pattern will match? The later pattern, `/app/dir/*.jsp`, will match because it is longer (has more characters) than `/**/*.jsp`. Note that this rule is not specified in any high-level interface for matching paths to request handlers, but it is an implementation detail.

Mapping Strategies

The `HandlerMapping` interface (shown in Listing 5-6) doesn't specify exactly how the mapping of request to handler is to take place, leaving the possible strategies wide open.

Listing 5-6. *HandlerMapping Interface*

```
package org.springframework.web.servlet;

public interface HandlerMapping {
  HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;
}
```

As you can see in Listing 5-6, a `HandlerMapping` returns not a `HandlerAdapter`, but a `HandlerExecutionChain`. This object encapsulates the handler object along with all handler interceptors for this request. The `HandlerExecutionChain` is a simple object and is used only between the `DispatcherServlet` and implementations of `HandlerMapping`. If you are not implementing your own custom `HandlerMapping`, then this object will be hidden from you.

Out of the box, Spring MVC provides three different mappers, all based on URLs. However, mapping is not tied to URLs, so feel free to use other mechanisms such as session state to decide on which request handler shall handle an incoming request.

BeanNameUrlHandlerMapping

The default strategy for mapping requests to handlers is the `org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping` class. This class treats any bean with a name or alias that starts with the `/` character as a potential request handler. The bean name, or alias, is then matched against incoming request URLs using Ant-style path matching. Listing 5-7 provides an example bean definition with a bean name containing a URL path.

Listing 5-7. *A Controller Mapped by a Bean Name*

```
  <bean name="/home"
    class="com.apress.expertspringmvc.flight.web.HomeController">
    <property name="flightService" ref="flightService" />
  </bean>
```

Caution You may not use a bean definition's `id` attribute to specify URL paths, because the XML specification forbids the `/` character in XML `id`s. You can, however, have both an `id` attribute and a name attribute on a single bean definition.

Path Components

Now how does that mapping translate to a full URI used by a client? Many paths are at work here, including the web application's context path, the servlet's mapped path, and then this `Controller`'s mapped path. How are they all combined and parsed when mapping a request to a handler?

By default, the path provided in the bean definition is inside the servlet's URL path. The servlet, in this case, is the `DispatcherServlet` that is declared and configured in the `web.xml`. For example, in Listing 5-8 we have mapped the `DispatcherServlet` to handle all requests for `/app/*`.

Listing 5-8. *Example DispatcherServlet Configuration*

```
<servlet>
  <servlet-name>spring</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>spring</servlet-name>
  <url-pattern>/app/*</url-pattern>
</servlet-mapping>
```

Therefore, to access the `HomeController` with a bean name of `/home`, a client must use the following full URI: `http://example.org/servletcontext/app/home`. In this case, `servletcontext` is the name of the servlet context this application is currently deployed to.

The default behavior, to be relative to the servlet mapping, is useful and preferable because it is decoupled from the URL pattern used to map the `DispatcherServlet` (in this case, `/app/*`). If the pattern changes, the bean name mappings do not need to change.

If you wish to write bean name mappings that include the servlet path mapping, you may do so by setting `alwaysUseFullPath` to `true` on an instance of `BeanNameUrlHandlerMapping`. To do this, simply declare an instance of `BeanNameUrlHandlerMapping` in your `ApplicationContext`. See Listing 5-9.

Listing 5-9. *Setting alwaysUseFullPath to True*

```
<bean
  class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
  <property name="alwaysUseFullPath" value="true" />
</bean>

<bean name="/app/home"
  class="com.apress.expertspringmvc.flight.web.HomeController">
  <property name="flightService" ref="flightService" />
</bean>
```

Multiple Mappings per Handler

You may also assign multiple mappings to a single handler. Simply separate each mapping with one or more spaces inside the bean name attribute, as shown in Listing 5-10.

Listing 5-10. *Multiple Mappings for a Single Handler*

```
<bean name="/home /homepage /index"
  class="com.apress.expertspringmvc.flight.web.HomeController">
  <property name="flightService" ref="flightService" />
</bean>
```

Because wildcards are supported, the example shown in Listing 5-10 could be shortened to Listing 5-11.

Listing 5-11. *Multiple Mappings with Wildcards*

```
<bean name="/home* /index"
  class="com.apress.expertspringmvc.flight.web.HomeController">
  <property name="flightService" ref="flightService" />
</bean>
```

Tip This convenient mapping technique is useful with `MultiActionControllers`, which are controllers that handle multiple URIs with separate methods. For more information, see the section "MultiActionControllers" in Chapter 6.

Default Mapping

It is also possible to set a default handler, in the case that no other mapping can satisfy the request. If you wish to designate a default handler, simply map that handler with `/*`, as shown in Listing 5-12.

Listing 5-12. *Setting a Controller As the Default Handler*

```
<bean name="/*"
  class="com.apress.expertspringmvc.flight.web.HomeController">
  <property name="flightService" ref="flightService" />
</bean>
```

The order in which you define your beans and mappings does not matter to the `BeanNameUrlHandlerMapping`. It attempts to find the best match at request time.

Match Algorithm

As you can see, there are many different strategies for mapping a request handler with a URL path. The `AbstractUrlHandlerMapping` class uses the following algorithm when performing a match.

1. Attempt an exact match. If found, exit from search.
2. Search all registered paths for a match. The most specific (longest) path pattern will win.
3. If no matches are found, use the default mapping (`/*`) if present.

BeanNameUrlHandlerMapping Shortcomings

While it is very convenient, there are shortcomings with `BeanNameUrlHandlerMapping`. This implementation of `HandlerMapping` is not able to map to prototype beans. In other words, all request handlers must be singletons when using `BeanNameUrlHandlerMapping`. Normally, `Controllers` are built as singletons, so this doesn't become an issue. However, as we'll see in the chapter covering `Controllers`, there are a few types of controllers that are indeed prototypes.

■**Note** Prototype beans are non-singleton beans. A new bean instance is created for every call to `getBean()` on the `ApplicationContext`. For more information, consult *Pro Spring*.

The `BeanNameUrlHandlerMapping` has another problem when your application begins to integrate interceptors. Because there is no explicit binding between this handler mapping and the beans it is mapping, it is impossible to create complex relationships between controllers and interceptors. We will cover interceptors in detail in Chapter 6.

If more complex handler mapping requirements arise, you may use the `SimpleUrlHandlerMapping` along with `BeanNameUrlHandlerMapping`.

SimpleUrlHandlerMapping

Created as an alternative to the simple `BeanNameUrlHandlerMapping`, the `SimpleUrlHandlerMapping` addresses the former's two shortcomings. It is able to map to prototype request handlers, and it allows you to create complex mappings between handlers and interceptors.

The path matching algorithms default to the same mechanism as `BeanNameUrlHandlerMapping`, so the patterns used to map URLs to request handlers remains the same.

To use a `SimpleUrlHandlerMapping`, simply declare it inside your `ApplicationContext`. The `DispatcherServlet` will recognize it by type, and it will not create an instance of `BeanNameUrlHandlerMapping`. This means that if you wish to use both mapping strategies, you must declare both in your `ApplicationContext`.

Tip The `DispatcherServlet` will chain handler mapping strategies, allowing you to mix and match as you see fit. Handler mappings also implement the `Ordered` interface.

To begin, we will port the previous mapping to use a `SimpleUrlHandlerMapping`, as shown in Listing 5-13.

Listing 5-13. *SimpleUrlHandlerMapping Example*

```
<bean
  class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="urlMap">
    <map>
      <entry key="/home" value-ref="homeController" />
    </map>
  </property>
</bean>

<bean id="homeController"
  class="com.apress.expertspringmvc.flight.web.HomeController">
  <property name="flightService" ref="flightService" />
</bean>
```

Unfortunately, it is a bit more verbose when mapping two different URL patterns to the same request handler. You must create two different mappings, as shown in Listing 5-14.

Listing 5-14. *Two Mappings for One Controller with SimpleUrlHandlerMapping*

```
<bean
  class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="urlMap">
    <map>
      <entry key="/home*" value-ref="homeController" />
      <entry key="/index" value-ref="homeController" />
    </map>
  </property>
</bean>
```

As you can see, we are mapping a URL directly to a request handler instance (in this case, the singleton homeController). If your request handlers are prototypes, you may instead use the mappings property of SimpleUrlHandlerMapping. Using this property, the mapping is between a URL and a bean name (as a String), thus decoupling the mapping from the actual bean instance. Using the mappings property, you are able to map to prototype request handlers, as they are looked up every time a request enters the system. See Listing 5-15.

Listing 5-15. *Mapping URLs to Bean Names for Use with Prototype Handlers*

```
<bean
  class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/home">homeController</prop>
    </props>
  </property>
</bean>
```

The SimpleUrlHandlerMapping maps default handlers in the same way as the BeanNameUrlHandlerMapping. To set a request handler as the default handler, simply map it to the path /*.

One of the main reasons to use SimpleUrlHandlerMapping is to take advantage of interceptors. While you can configure interceptors with BeanNameUrlHandlerMapping, it is very difficult to create different combinations of handlers and interceptors. Using SimpleUrlHandlerMapping makes it easy to create custom handler chains per request handler. We will visit interceptors in Chapter 6.

Custom Mapping Strategy

The power and flexibility of Spring MVC's request mapping really shines when a non–URL-based mapping strategy is required. Because the HandlerMapping interface doesn't require a URL to be involved in the mapping, the possibilities are quite open.

To illustrate a mapping strategy not based on URLs, let's consider mapping requests to handlers based solely on request parameters.

To begin with, we will subclass AbstractHandlerMapping to take advantage of ordering, the ability to set a default handler, and other life cycle callbacks. The new RequestParameterHandlerMapping class (Listing 5-16) will map request parameter values from a specified parameter name to handler instances.

Listing 5-16. *RequestParameterHandlerMapping*

```
public class RequestParameterHandlerMapping extends AbstractHandlerMapping
        implements InitializingBean {

    public final static String DEFAULT_PARAM_NAME = "handler";
    private String parameterName = DEFAULT_PARAM_NAME;

    private final Map<String, Object> paramMappings =
        new HashMap<String, Object>();
    public final void setParamMappings(Map<String, Object> paramMappings) {
        this.paramMappings.putAll(paramMappings);
    }

    public final void setParameterName(String parameterName) {
        this.parameterName = parameterName;
    }

    @Override
    protected Object getHandlerInternal(HttpServletRequest request)
            throws Exception {
        String parameterValue = request.getParameter(parameterName);
        return paramMappings.get(parameterValue);
    }

    public void afterPropertiesSet() throws Exception {
        Assert.hasText(parameterName,
          "parameterName must not be null or blank");
    }

}
```

Because this class extends `AbstractHandlerMapping`, if no handler exists for the request parameter, then the `AbstractHandlerMapping` will attempt to load the default handler, which can be set via the XML bean definition. See Listing 5-17.

Listing 5-17. *RequestParameterHandlerMapping XML Definition*

```
<bean
  class="com.apress.expertspringmvc.chap5.RequestParameterHandlerMapping">
  <property name="defaultHandler" ref="defaultController" />
  <property name="parameterName" value="action" />
  <property name="paramMappings">
    <map>
      <entry key="load" value-ref="loadController" />
      <entry key="save" value-ref="saveController" />
    </map>
  </property>
</bean>
```

With this configuration, the URL `http://example.org/springapp/app?action=load` would be routed to the `loadController`. Remember that `/app` is just the `DispatcherServlet` mapping and not specific to any controller or request handler.

Summary

Mapping incoming requests to request handlers is very flexible in Spring MVC. Out of the box, URL mapping methods are provided, but it's very easy to create mapping strategies that use any information available in the `HttpServletRequest`.

The `BeanNameUrlHandlerMapping` is the default mapping strategy and is used if no other mapping strategies are defined in the `ApplicationContext`. This strategy, while simple and easy, does have a few limitations. If you require complex interceptor mapping or the use of prototype beans for handlers, you will need to use the `SimpleUrlHandlerMapping`.

Handler mapping strategies can be ordered using the `Ordered` interface, allowing you to utilize multiple methods to resolve incoming requests to handlers.

HandlerExceptionResolver

When an exception occurs from handling a request, Spring MVC can catch the exception for you and route the request to a particular error page or other exception handling code. The `HandlerExceptionResolver` will handle any exception thrown inside the `HandlerInterceptors`, the `Controllers`, or the `View` rendering. Typically, an exception is mapped to a particular error page, but it is easy to extend this functionality for your particular error handling needs.

By using a `HandlerExceptionResolver`, shown in Listing 5-18, it is easy to centralize error handling and configuration. Otherwise, each controller and interceptor would have to contain duplicate code and logic for each exception that could be thrown.

Listing 5-18. *HandlerExceptionResolver Interface*

```
package org.springframework.web.servlet;

public interface HandlerExceptionResolver {

    ModelAndView resolveException(
        HttpServletRequest request, HttpServletResponse response,
        Object handler, Exception ex);

}
```

The `DispatcherServlet` is configured by default to look for all beans in its `ApplicationContext` of type `HandlerExceptionResolver`. If it finds one or more, it will order them using the `org.springframework.core.Ordered` interface if the bean implements it. If no `HandlerExceptionResolver` is found, no exception resolving will take place. Of course, any mapped exceptions you have specified in the `web.xml` will still apply if the exception isn't handled by a `HandlerExceptionResolver`.

You may also tell the `DispatcherServlet` to use only a single exception resolver, ignoring all others that may be present in the `ApplicationContext`. Simply set the `detectAllHandlerExceptionResolvers` property of the `DispatcherServlet` to `false`, and then define a single bean with the name `handlerExceptionResolver`.

The default implementation of this interface is the `org.springframework.web.servlet.handler.SimpleMappingExceptionResolver`. This class maps exceptions to view names by the exception class name or a substring of the class name. This implementation can be configured for individual `Controllers` or for globally for all handlers. The example configuration in Listing 5-19 illustrates these options.

Listing 5-19. *Example SimpleMappingExceptionResolver ApplicationContext*

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="exceptionMapping"
    class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
      <props>
        <prop key="ApplicationException">appErrorView</prop>
        <prop key="SomeOtherException">someErrorView</prop>
        <prop key="java.lang.Exception">genericErrorView</prop>
      </props>
    </property>
  </bean>
</beans>
```

The `exceptionMappings` property is a `java.util.Properties` with substrings (explained later in this section) of exception class names as keys and `View` names as values. Notice how you can specify a fully qualified class name or only part of a class name for the key. The error `View` name will be ultimately resolved by a `ViewResolver`.

Listing 5-19 does not specify a particular request handler, so it will be applied to any mapped exception by any handler. However, you can bind an exception resolver to specific handlers to create very specific mappings. Use this technique when you require displaying different error pages for the same exception thrown by two different controllers.

■ **Caution** Mapping exceptions to individual handlers only works if the handler is a singleton. Unless you are using `ThrowawayControllers`, this should not be an issue because normally `Controllers` are singletons. However, it is always possible to run any controller as a prototype.

Listing 5-20 provides two examples, one for mapping an exception resolver to a single request handler, and the other for mapping to multiple request handlers.

Listing 5-20. *Two Exception Resolvers for Specific Handlers*

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC
```

```
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="exceptionMappingForSingleController"
    class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="mappedHandlers">
      <set>
        <ref bean="someController" />
      </set>
    </property>
    <property name="exceptionMappings">
      <props>
        <prop key="ApplicationException">appErrorView</prop>
        <prop key="SomeOtherException">someErrorView</prop>
        <prop key="java.lang.Exception">genericErrorView</prop>
      </props>
    </property>
  </bean>

  <bean id="exceptionMappingForMultipleControllers"
    class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="mappedHandlers">
      <set>
        <ref bean="anotherController" />
        <ref bean="mainController" />
      </set>
    </property>
    <property name="exceptionMappings">
      <props>
        <prop key="java.lang.Exception">differentErrorView</prop>
      </props>
    </property>
  </bean>
</beans>
```

Two exception resolvers are defined in the `ApplicationContext` in Listing 5-20. Each defines a set of mapped handlers that will define when the exception resolver is applied. If an exception resolver encounters an exception from a handler it is not mapped to, it will simply ignore the exception. Note that this behavior is only for exception resolvers that are mapped to at least one handler.

You can control the order in which the `DispatcherServlet` will call each exception resolver. To do this, simply add a property named `order` and set it to a positive integer. Any exception resolvers not specified with an order will be randomly placed at the end of the ordered list.

Listing 5-21 provides an example of setting an order priority.

Listing 5-21. *Example of Ordered Exception Resolver*

```
<bean id="anotherExceptionMapping"
  class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
  <property name="order" value="1" />
  <property name="mappedHandlers">
    <set>
```

Pattern Matching Rules

The rules for pattern matching of the exception names might not be obvious, so we cover them here. There are two rules to be aware of.

Rule Number One

The first rule to be aware of is that the shorter the pattern string, the higher priority it will receive. For example, the mapping `Excep` will match any exception whose class name contains that substring (in other words, nearly every exception). Even if you have another mapping with the exact class name, the shorter `Excep` mapping will resolve first. Listings 5-22 through 5-24 illustrate this.

Listing 5-22 contains the two exception classes we'll use for this example. It is a simple class hierarchy, with `ExceptionChild` subclassing `ExceptionParent`.

Listing 5-22. *Example Exception Classes*

```
public class ExceptionParent extends Exception { }
public class ExceptionChild extends ExceptionParent { }
```

Listing 5-23 illustrates an exception resolver with two mappings. The first is a very general mapping, handling any exception whose name includes the substring `Excep`. The second mapping is an explicit mapping for an exception whose names includes the fuller `ExceptionChild`.

Listing 5-23. *Exception Mapping Example*

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
 <bean id="exceptionMapping"
  class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
  <property name="exceptionMappings">
    <props>
      <prop key="Excep">exceptionPage</prop>
      <prop key="ExceptionChild">moreSpecificPage</prop>
    </props>
  </property>
 </bean>
</beans>
```

Listing 5-24 contains sample code that programmatically illustrates what view name is resolved when an ExceptionChild exception is thrown.

Listing 5-24. *Test Case*

```
ModelAndView mav = resolver.resolveException(req, res, handler,
    new ExceptionChild());
assertEquals("exceptionPage", mav.getViewName());   // true!
```

Notice how, in Listing 5-24, the SimpleMappingExceptionResolver returned the view name exceptionPage, even though there was a rule to map an ExceptionChild exception.

Rule Number Two

The second rule comes in two parts. Exception mappings are aware of their superclasses, so a mapping for a class will resolve to that class and all of its subclasses. Given the exception classes from the previous example, the code in Listing 5-25 illustrates this rule.

Listing 5-25. *Exception Resolver Configuration for ExceptionParent*

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
 <bean id="exceptionMapping"
  class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
  <property name="exceptionMappings">
    <props>
      <prop key="ExceptionParent">parentPage</prop>
    </props>
  </property>
 </bean>
</beans>
```

Listing 5-26 simply shows that even those an ExceptionChild was thrown, the mapping for ExceptionParent resolves.

Listing 5-26. *Test Case*

```
ModelAndView mav = resolver.resolveException(req, res, handler,
    new ExceptionChild());  // throwing child subclass
assertEquals("parentPage", mav.getViewName());   // true!
```

Now, here is the second part of the rule. If you specify both the parent exception and the child exception, then the child exception will resolve. So, even though the resolving logic will scan the exception class hierarchy for a match, it will prefer a match lower in the tree.

Listing 5-27 contains a simple mapping with both `ExceptionParent` and `ExceptionChild`. Which one will resolve is based on which exception is thrown. Listing 5-28 shows that when throwing `ExceptionChild`, the view name `childPage` resolves because `ExceptionChild` is more specific than `ExceptionParent`.

Listing 5-27. *Exception Resolver Mapping Both ExceptionParent and ExceptionChild*

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
 <bean id="exceptionMapping"
  class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
  <property name="exceptionMappings">
    <props>
      <prop key="ExceptionParent">parentPage</prop>
      <prop key="ExceptionChild">childPage</prop>
    </props>
  </property>
 </bean>
</beans>
```

Listing 5-28. *Test Case*

```
ModelAndView mav = resolver.resolveException(req, res, handler,
    new ExceptionChild());  // throwing child subclass
assertEquals("childPage", mav.getViewName());   // true!
```

Of course, if you configured that last exception resolver with a mapping of `Exce`, then that would take precedence over either previous mapping.

Summary

To summarize, the `HandlerExceptionResolver` interface provides a mechanism to centralize exception handling and remove it from the primary work flow logic. You can configure multiple exception resolvers in an `ApplicationContext`, and they can be ordered by priority.

Spring provides a single implementation of this interface called `SimpleMappingExceptionResolver` that maps exception names to error pages. This implementation can match full class names or substrings, will prefer a shorter name, and is aware of the class hierarchy when attempting to match the exception.

The `DispatcherServlet` is aware of all exception resolvers in the `ApplicationContext` and can order them based on priority. You can change this behavior by setting the `DispatcherServlet`'s `detectAllHandlerExceptionResolvers` property to `false`, in which case you will need to define a single exception resolver with the name `handlerExceptionResolver`.

LocaleResolver

The `org.springframework.web.servlet.LocaleResolver` is a Strategy interface for retrieving and setting a `java.util.Locale` during a web request. The Gang of Four, authors of *Design Patterns* (Addison Wesley, 1995), write this of the Strategy pattern: "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."

■ **Tip** The Strategy pattern is heavily implemented within the Spring Framework. One of the reasons this pattern is preferred is because class inheritance is generally avoided as a means to share behavior. The Strategy pattern allows you to share behavior much more easily.

The `LocaleResolver` defines the contract of `Locale` resolution and modification. It leaves the details of these methods up to implementations. Most importantly, the implementations can be exchanged freely without affecting the system.

The `Locale` is mostly used when the application needs to display translated text for the user interface, although it is also useful for formatting numbers and currencies. It assists the general internationalization (i18n) features of the Java platform (`http://java.sun.com/docs/books/tutorial/i18n`) and Spring MVC for providing language and culture independent applications. You will see many examples of this in Chapter 7, which covers user interface options. For now, it's important to know that the `Locale` is set per user and intended to be accessible to both the work flow and the user interface.

By default, generic Java web applications will respect the Accept-Language (`http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.4`) header of a HTTP request. The servlet container will map the languages specified in the header into `Locale` objects and set the primary choice as the chosen `Locale`.

■ **Tip** For more information on how Java web applications handle this header, see section 4.8 of the Servlet 2.4 Specification (`http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html`) or `HttpServletRequest`'s `getLocale()` and `getLocales()` methods.

Many applications, however, require more control when selecting the user's `Locale` or modifying it. This strategy interface allows you to customize where the `Locale` information comes from and how to change it. First, let's look at the `LocaleResolver` interface, shown in Listing 5-29 and Figure 5-2.

Listing 5-29. *LocaleResolver Interface*

```
package org.springframework.web.servlet;

public interface LocaleResolver {
    Locale resolveLocale(HttpServletRequest request);

    void setLocale(HttpServletRequest request, HttpServletResponse response,
           Locale locale);
}
```

■ **Tip** To clear out the locale using this strategy interface, simply set the `locale` property of the `LocaleResolver` implementation to `null`.

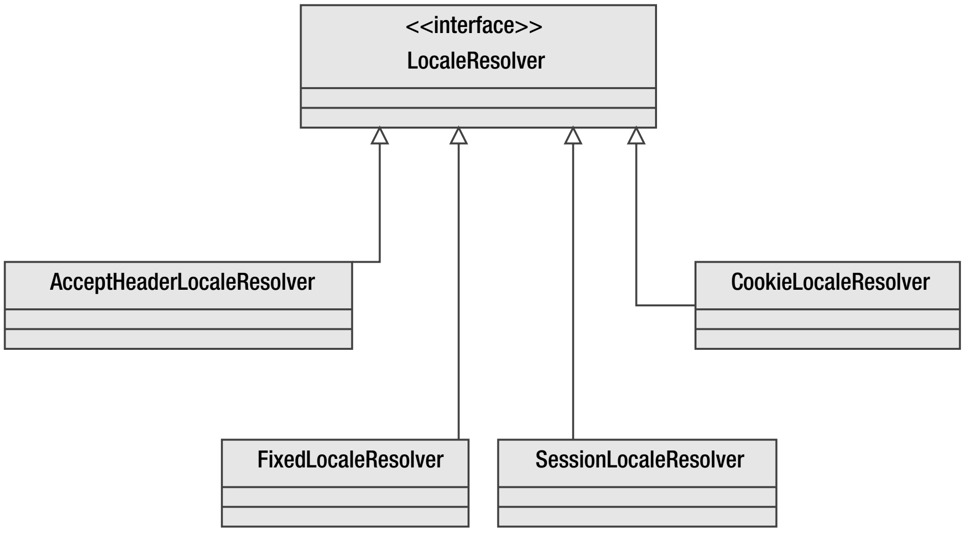

Figure 5-2. *LocaleResolver class hierarchy*

Once you have chosen an appropriate `LocaleResolver`, you need to register it with the `DispatcherServlet` by creating a bean definition with the name `localeResolver` in your `ApplicationContext`. The `DispatcherServlet` will look for that name before falling back to its default implementation, the `AcceptHeaderLocaleResolver`.

The `LocaleResolver` interface defines how to query and set a `Locale`, but it does not define when or how you should use it. Spring MVC's i18n infrastructure (discussed in Chapters 7 and 9) will read out the `Locale` when performing translations and culture-specific formatting. This work is typically done under the hood, so your job as a developer is to set the locale via the `LocaleResolver`.

This action is often a manual operation, helped tremendously by the chosen strategy, because many users and clients cannot be trusted to set their browser's language preferences correctly. In other words, you probably can't trust that the Accept-Language HTTP header is correctly configured all the time.

■ **Note** The `DispatcherServlet` does not support chaining `LocaleResolver`s, so you are allowed to choose only one implementation.

Setting a Locale

Your application's design will dictate when you should call `setLocale`, normally in response to some user action. For example, the application might provide a page with language choices, and the user will be able to choose one and submit it back to the server.

In one possible implementation, you would create a `Controller` that delegates to a `LocaleResolver` in order to store the user's chosen `Locale`. Listings 5-30 and 5-31 show you how to do this.

Listing 5-30. *HTML Form, Choosing Language*

```
<form action="setLocale" method="post">
  <p>
  Language: <select name="language">
            <option value="en">English</option>
            <option value="de">German</option>
            </select>
  </p>
  <p>
  <input type="submit" />
  </p>
</form>
```

Listing 5-31. *SetLocaleController*

```
public class SetLocaleController extends AbstractController {

  @Override
  protected ModelAndView handleRequestInternal(HttpServletRequest req,
            HttpServletResponse res) throws Exception {
    String language = req.getParameter("language");
    Locale locale = StringUtils.parseLocaleString(language);

    // How did we get a reference to the localeResolver?
    // See Listings 5-32 and 5-34 for the two strategies
    // for obtaining a reference
    // to this localeResolver instance
    localeResolver.setLocale(req, res, locale);

    return new ModelAndView("setLocaleSuccess");
  }

}
```

■**Tip** Spring MVC provides a `LocaleChangeInterceptor` that performs the exact same operation as the above example `Controller`. This interceptor is recommended, especially if many forms all have the same locale request parameters.

Retrieving a LocaleResolver

Clearly you may use the `LocaleResolver` to set the locale, but what is the best way to obtain a reference to the `localeResolver` from inside the `Controller`? There are at least two ways to get the `LocaleResolver` object, each with different advantages. Which way you choose will be up to you.

The most obvious way to get the `LocaleResolver` is to rely on Dependency Injection. The `LocaleResolver` instance is a bean in the `ApplicationContext` like all other objects in the system, so Spring will be happy to set this object into your `Controller` via DI for you. The `DispatcherServlet` will recognize only one `LocaleResolver` in the `ApplicationContext` (it won't chain multiple resolvers of this type), so typically there will be only one instance in the application. If you have defined a `LocaleResolver` in the context, consider altering your `Controller` to allow for injection of the resource, as shown in Listings 5-32 and 5-33.

Listing 5-32. *Adding a Setter Method for Dependency Injection*

```
  private LocaleResolver localeResolver;

  public void setLocaleResolver(LocaleResolver localeResolver) {
    this.localeResolver = localeResolver;
  }
```

Listing 5-33. *SetLocaleController ApplicationContext*

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

<beans>

  <bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />
  <bean id="setLocaleController"
    class="com.apress.expertspringmvc.chap4.SetLocaleController">
    <property name="localeResolver" ref="localeResolver" />
  </bean>

</beans>
```

There are situations where simple Dependency Injection won't work, such as when Spring doesn't manage the life cycle of the handler object. For instance, the `HandlerAdapter` interface was created to allow any request handler to be easily integrated into the `DispatcherServlet`. In cases such as these, the `DispatcherServlet` places some of the framework objects, such as the `LocaleResolver`, into the servlet request as request scoped attributes. This allows for any object handling the `HttpServlerRequest` to be able to access the `LocaleResolver`.

To conveniently access the resolver without Dependency Injection, use the `org.springframework.web.servlet.support.RequestContextUtils` class and its helpful `getLocaleResolver()` method. This utility method encapsulates the knowledge of where the `LocaleResolver` is placed in the request scope, making retrieval safer for the client. The `Controller` can be modified as shown in Listing 5-34.

Listing 5-34. *RequestContextUtils.getLocaleResolver Example*

```
  @Override
  protected ModelAndView handleRequestInternal(HttpServletRequest req,
            HttpServletResponse res) throws Exception {
    String language = req.getParameter("language");
    Locale locale = StringUtils.parseLocaleString(language);
    LocaleResolver localeResolver = RequestContextUtils.getLocaleResolver(req);
    localeResolver.setLocale(req, res, locale);
    return new ModelAndView("setLocaleSuccess");
  }
```

This class no longer needs the `setLocaleResolver()` method, as Dependency Injection is no longer used.

Which method should you use? Using Dependency Injection is always easier to test than using `RequestContextUtils`, so the DI solution is preferable.

You have now seen the `LocaleResolver` interface and the different options of interacting with it. How the `Locale`, once it is set, affects the i18n features will be covered in Chapters 7 and 9. It's time now to look in detail at the different implementations of `LocaleResolver`.

AcceptHeaderLocaleResolver

The `DispatcherServlet` will default to the `org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver` class if no other `LocaleResolvers` are specified in the `ApplicationContext`. This implementation simply delegates to the `HttpServletRequest`'s `getLocale()` method, thus obeying the Accept-Language HTTP header. Because the header originates from the client, there is no way to change the `Locale`, so the `AcceptHeaderLocaleResolver` will throw an exception if `setLocale()` is called.

There is no need to specify this class in your `ApplicationContext`, as the `DispatcherServlet` will create it if no implementations can be found.

FixedLocaleResolver

The simplest implementation is the FixedLocaleResolver. This class allows you to define the Locale in the ApplicationContext, and because the Locale choice is fixed, it's one size fits all with this Strategy. This is a useful and easy way to override and ignore any client's language choice, for example. Simply define an instance of this class in your ApplicationContext with the bean name localeResolver, and the DispatcherServlet will recognize it automatically.

Listing 5-35 contains an example configuration of a FixedLocaleResolver.

Listing 5-35. *Sample FixedLocaleResolver*

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

<beans>

  <bean id="fixedLocaleResolver"
    class="org.springframework.web.servlet.i18n.FixedLocaleResolver">
    <property name="defaultLocale" value="en" />
  </bean>

</beans>
```

Using the configuration from Listing 5-35, no matter what the client advertises via its headers, the framework will force the locale to be en.

You might notice that the defaultLocale property is of type java.util.Locale, yet all we specified in the above configuration is the string en. Spring will use its LocaleEditor property editor to convert from the string to a full Locale instance. LocaleEditor is one of the many property editors that Spring will create and register by default.

■ **Tip** Spring's use of PropertyEditors from the Java Bean specification is quite extensive. Take the time to learn what editors Spring provides out of the box (a subset is covered in Chapter 6).

If your applications require providing each user their own unique experience, the FixedLocaleResolver falls quite short. With the default class AcceptHeaderLocaleResolver (while it does provide a personalized experience) it's impossible for the application to change its value. There are two other implementations of LocaleResolver that allow the application to change the Locale for the user, the CookieLocaleResolver and the SessionLocaleResolver. Both of these implementations support changing and storing the Locale across requests.

CookieLocaleResolver

The CookieLocaleResolver sets and retrieves the Locale object via a browser cookie. This strategy is useful when the application does not support sessions and the state must be kept client side.

Simply declare this class in your ApplicationContext to use it. Note that you can configure the name of the cookie if you choose, but the class provides a sensible default. If you wish to clear the Locale cookie, simply call setLocale() and pass in a null locale.

Listing 5-36 contains a sample bean definition for a CookieLocaleResolver.

Listing 5-36. *CookieLocaleResolver Bean Definition*

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

<beans>

  <bean id="cookieLocaleResolver"
    class="org.springframework.web.servlet.i18n.CookieLocaleResolver" />

</beans>
```

■**Note** If there is no Locale cookie present, this class will fall back to ServletRequest's getLocale() method. The getLocale() method returns the client's preferred Locale, as dictated by the Accept-Language HTTP header. If the client did not specify an Accept-Language header, the method returns the default Locale of the server.

SessionLocaleResolver

The SessionLocaleResolver stores the user's Locale inside the HttpSession object, and it supports both retrieval and modification. It offers a nice alternative to storing the locale state in a cookie. As with the CookieLocaleResolver, if no Locale is found in the session, this class will fall back to the getLocale() method of HttpServletRequest.

This implementation (see Listing 5-37) is as easy to declare as the CookieLocaleResolver.

Listing 5-37. *SessionLocaleResolver Bean*

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

<beans>
```

```
  <bean id="sessionLocaleResolver"
    class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />

</beans>
```

Summary

So how do you choose which locale management strategy to use? It all depends on what your application requirements are, and as usual, Spring doesn't force one decision for you. In fact, if your needs aren't covered by the included strategy implementations, the `LocaleResolver` interface is simple enough to create a customization if required.

Tip Never feel constrained by the provided solutions and implementations. More often than not, there's an interface or abstract class for easy customization.

If your application doesn't allow your users to change their `Locale`, but you do want to acknowledge their browser's defaults, then stick with the default `AcceptHeaderLocaleResolver`. This strategy emulates the Servlet specification's default behavior and requires no configuration. It's also the option of least surprise, as it performs as most people would expect.

If you need to force a particular `Locale` and it can never be changed, then `FixedLocaleResolver` is the one for you. This class is very simple to set up, but is quite limiting.

When your application requires `Locales` to be changed by the user from inside the web application, then the `CookieLocaleResolver` and the `SessionLocaleResolver` are your two choices. If your application is already using sessions, then the `SessionLocaleResolver` is a logical choice. However, if you require the `Locale` choice to persist longer than the session, then the `CookieLocaleResolver` is your only choice. There is no clear winner here, so choose the option that best fits your situation.

MultipartResolver

Handling file uploads is a standard feature of web frameworks, and Spring MVC's `org.springframework.web.multipart.MultipartResolver` provides the strategy interface for this functionality. Like many other features, Spring doesn't reinvent the wheel when it comes to file upload handling. Out of the box, Spring provides two implementations of `MultipartResolver`, one for Jakarta Commons' FileUpload (`http://jakarta.apache.org/commons/fileupload`) and one for Jason Hunter's COS (`http://www.servlets.com/cos`).

Tip COS stands for `com.oreilly.servlet`, as the library was originally written for Jason Hunter's *Java Serlvet Programming* (O'Reilly, 2001).

HTTP file uploading, or "Form-based File Upload in HTML," is defined in RFC 1867 (`http://www.ietf.org/rfc/rfc1867.txt`). By creating an HTML input field of `type="file"` and setting the form's `enctype="multipart/form-data"`, the browser can send a text or binary file to the server as part of a HTTP POST request.

The `DispatcherServlet` will look for a single bean in the `ApplicationContext` with the name `multipartResolver`. If one is found, it will pass each incoming request to the resolver in order to wrap the `HttpServletRequest` with a subclass that can expose the file upload. If a multipart resolver is not located, then no multipart file handling will be available. Note that the `DispatcherServlet` does not chain `MultipartResolvers`.

Unlike previous resolvers such as `LocaleResolver`, client code is never intended to interact with this interface directly. The `DispatcherServlet` manages the work flow of the `MultipartResolver`, and the client code will simply cast the request object to an `org.springframework.web.multipart.MultipartHttpServletRequest` wrapper object in order to get the uploaded file(s).

Listing 5-38 contains the `MultipartResolver` interface.

Listing 5-38. *MultipartResolver Interface*

```
package org.springframework.web.multipart;

public interface MultipartResolver {

  boolean isMultipart(HttpServletRequest request);

  MultipartHttpServletRequest resolveMultipart(HttpServletRequest request)
    throws MultipartException;

  void cleanupMultipart(MultipartHttpServletRequest request);

}
```

The `isMultipart()` method is called by the `DispatcherServlet` in order to determine if the incoming request is a multipart request. The implementation will most likely check the Content-Type of the request for a value of `multipart/form-data`, but this can be only part of the heuristics.

If the request does indeed contain uploaded files, the `resolveMultipart()` method is called, returning the `MultipartHttpServletRequest` (see Listing 5-39). This wrapping object adds methods to retrieve the uploaded files.

■ **Caution** The `MultipartResolver` will only wrap requests with a `MultipartHttpServletRequest` if the request actually contains file uploads.

Listing 5-39. *MultipartHttpServletRequest Interface*

```
package org.springframework.web.multipart;

public interface MultipartHttpServletRequest extends HttpServletRequest {

  Iterator getFileNames();

  MultipartFile getFile(String name);

  Map getFileMap();

}
```

Before the end of the request life cycle and after the handling code has had a chance to work with the uploaded files, the `DispatcherServlet` will then call `cleanupMultipart()`. This removes any state left behind by the file upload implementation code, such as temporary files on the file system. Therefore, it is important that any request handling code should work with the uploaded files before request processing finishes.

So which library should you use, Commons' FileUpload or COS? The choice is up to you, as both have been around for years and are considered stable. However, keep in mind that Commons' FileUpload will probably receive more maintenance in the future. Of course, if neither provides the features you require, you may implement a new `MultipartResolver`.

Example

Working with file uploads is actually quite simple, as most of the mechanisms are handled by the `DispatcherServlet` and thus hidden from request handling code. For an example, we will register a Jakarta Commons FileUpload `MultipartResolver` and create a `Controller` that saves uploaded files to a temporary directory.

Listing 5-40 contains the configuration required for the `CommonsMultipartResolver`.

Listing 5-40. *MultipartResolver ApplicationContext*

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

<beans>

  <bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="maxUploadSize" value="2000000" />
  </bean>

  <bean name="/handleUpload"
    class="com.apress.expertspringmvc.chap4.HandleUploadController">
    <property name="tempDirectory" value="/tmp" />
  </bean>

</beans>
```

Note that we declared the multipart resolver in the same ApplicationContext as our Controller. We recommend grouping all web-related beans in the same context.

Next, we create the form for the file upload, as shown in Listing 5-41.

■ **Tip** It's very important to set the enctype attribute of the <form> element to multipart/form-data.

Listing 5-41. *HTML File Upload Form*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>File Upload Form</title>
</head>
<body>

<form action="spring/handleUpload" method="post" enctype="multipart/form-data">

  File: <input type="file" name="uploaded" />

  <input type="submit" />

</form>
</body>
</html>
```

The Controller that handles the request is shown in Listing 5-42. Notice how it must cast the request object to a MultipartHttpServletRequest before extracting the file. The utility class FileCopyUtils, provided by Spring, contains convenience methods such as copying an input stream to an output stream.

Listing 5-42. *File Upload Controller*

```
public class HandleUploadController extends AbstractController
    implements InitializingBean {

  private File destinationDir;

  public void setDestinationDir(File destinationDir) {
    this.destinationDir = destinationDir;
  }
  public void afterPropertiesSet() throws Exception {
    if (destinationDir == null) {
      throw new IllegalArgumentException("Must specify destinationDir");
    } else if (!destinationDir.isDirectory() && !destinationDir.mkdir()) {
      throw new IllegalArgumentException(destinationDir + " is not a " +
        "directory, or it couldn't be created");
    }
  }

  protected ModelAndView handleRequestInternal(HttpServletRequest req,
          HttpServletResponse res) throws Exception {
        res.setContentType("text/plain");

        if (! (req instanceof MultipartHttpServletRequest)) {
            res.sendError(HttpServletResponse.SC_BAD_REQUEST,
                "Expected multipart request");
            return null;
        }

        MultipartHttpServletRequest multipartRequest =
            (MultipartHttpServletRequest) req;
        MultipartFile file = multipartRequest.getFile("uploaded");
        File destination = File.createTempFile("file", "uploaded",
                destinationDir);
        FileCopyUtils.copy(file.getInputStream(),
                new FileOutputStream(destination));

        res.getWriter().write("Success, wrote to " + destination);
        res.flushBuffer();
        return null;
  }

}
```

If you are creating command beans (see `BaseCommandController` and `SimpleFormController` in Chapter 6) to encapsulate the request parameters from forms, you can even populate a property of your command object from the contents of the uploaded file. In other words, instead of performing the manual operation of extracting the file from the `MultipartFile` instance (as we did in the preceding example in Listing 5-42), Spring MVC can inject the contents of the uploaded file (as a `MultipartFile`, `byte[]`, or `String`) directly into a property on your command bean. With this technique there is no need to cast the `ServletRequest` object or manually retrieve the file contents.

We'll cover binding request parameters from forms in the next chapter, so we won't jump ahead here and confuse the topic at hand. But we will provide the hint required to make the file contents transparently show up in your command bean: you must register either `ByteArrayMultipartFileEditor` or `StringMultipartFileEditor` with your data binder (for instance, inside the `initBinder()` method of your form controller). What does that mean? Hang tight, or skip to Chapter 7.

As long as the contents of the uploaded file aren't too large, we recommend the direct property binding because it is less work for you and certainly more transparent.

ThemeResolver

Spring MVC supports a concept of *themes*, which are interchangeable looks and feels for your web application. Often called *skins*, themes are a way to abstract a look and feel (color scheme, logo, size of buttons, and so on) from the user interface. This is helpful to the user interface implementer, because the skin information can be rendered at runtime, instead of simply duplicating each page once for each look and feel. We will cover themes in greater detail in the Chapter 7. For now, we will focus on how to choose and manipulate themes for each user's requests. You will find that the concepts here are very similar to the `LocaleResolver`.

Listing 5-43 contains the `ThemeResolver` interface.

Listing 5-43. *ThemeResolver Interface*

```
package org.springframework.web.servlet;

public interface ThemeResolver {
  String resolveThemeName(HttpServletRequest request);

  void setThemeName(HttpServletRequest request, HttpServletResponse response,
    String themeName);
}
```

As you can see, the `ThemeResolver` interface resembles the `LocaleResolver` interface very closely. One major difference between the two is `ThemeResolver` returns strings instead of a strongly typed objects. The resolution of the theme name to a `org.springframework.ui.context.Theme` object is done via an `org.springframework.ui.context.ThemeSource` implementation.

The `ThemeResolver` interface has the same types of implementations as the `LocaleResolver` interface. Out of the box, Spring MVC provides a `FixedThemeResolver`, a `CookieThemeResolver`, and a `SessionThemeResolver`. Just like their `LocaleResolver` counterparts, both `CookieThemeResolver` and `SessionThemeResolver` support retrieving and changing the theme, while `FixedThemeResolver` only supports a read-only theme.

Figure 5-3 illustrates the class hierarchy for the ThemeResolver and its subclasses.

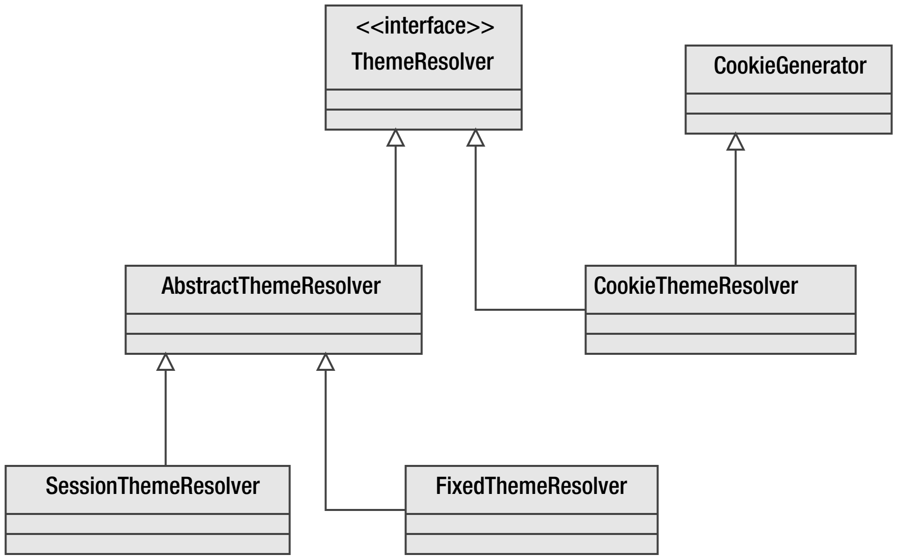

Figure 5-3. *ThemeResolver class hierarchy*

The DispatcherServlet does not support chaining of ThemeResolvers. It will simply attempt to find a bean in the ApplicationContext with the name themeResolver. If no ThemeResolvers are located, then the DispatcherServlet will create its own FixedThemeResolver configured only with the defaults.

Working with the configured ThemeResolver is no different than working with the LocaleResolver. The DispatcherServlet places the ThemeResolver into each request as an HttpServletRequest attribute. You then access this object through the RequestContextUtils utility class and its getThemeResolver() method.

Summary

A theme is a skin, or look and feel, for your web application that is easily changed by the user or application. The ThemeResolver interface encapsulates the strategy for reading and setting the theme for a user's request. Similar to the LocaleResolver, the ThemeResolver supports a fixed theme, or storing the theme in a cookie or in the HttpSession object.

The DispatcherServlet will look for a bean with the name themeResolver in the ApplicationContext upon startup. If it does not find one, it will use the default FixedThemeResolver.

We'll discuss themes in detail in Chapter 7. For now, it's important to know that there is one for each DispatcherServlet and the default, if none are specified, is the FixedThemeResolver.

Summary

Spring MVC has a full-featured processing pipeline, but through the use of sensible abstractions and extensions, it can be easily extended and customized to create powerful applications. The key is the many interfaces and abstract base classes provided for nearly every step along the request's life cycle.

As a developer, you are encouraged to implement and extend the provided interfaces and implementations to customize your users' experiences. Don't be constrained by the provided implementations. If you don't see something you need, chances are it's very easy to create.

For more information on themes and views, including the `ViewResolver`, continue on to Chapter 7. For more information on `Controllers` (Spring MVC's default request handlers) and interceptors, let's now continue on to Chapter 6.