(/)

web.xml vs Initializer with Spring

Last modified: April 2, 2018

by baeldung (https://www.baeldung.com/author/baeldung/)

Spring (https://www.baeldung.com/category/spring/) +

Servlet (https://www.baeldung.com/tag/servlet/)

1. Overview

In this article we'll cover three different approaches of configuring a *DispatcherServlet* available in recent versions of the *Spring Framework:*

1. We'll start with an *XML* configuration and a *web.xml* file
2. Then we'll migrate the Servlet declaration from the *web.xml* file to Java config, but we'll leave any other configuration in *XML*
3. Finally in the third and final step of the refactoring, we'll have a 100% Java-configured project

2. The *DispatcherServlet*

One of the core concepts of *Spring MVC* is the *DispatcherServlet*. The Spring documentation (http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html) defines it as:

> A central dispatcher for HTTP request handlers/controllers, e.g. for web UI controllers or HTTP-based remote service exporters. Dispatches to registered handlers for processing a web request, providing convenient mapping and exception handling facilities.

Basically the *DispatcherServlet* is the entry point of every *Spring MVC* application. Its purpose is to intercept *HTTP* requests and to dispatch them to the right component that will know how to handle it.

3. Configuration with *web.xml*

If you deal with legacy *Spring* projects it is very common to find *XML* configuration and until *Spring* 3.1 the only way to configure the *DispatcherServlet* was with the *WEB-INF/web.xml* file. In this case there are two steps required.

Let's see an example configuration – the first step is the Servlet declaration:

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/dispatcher-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

With this block of *XML* we are declaring a servlet that:

1. Is named "*dispatcher*"
2. Is an instance of *org.springframework.web.servlet.DispatcherServlet*
3. Will be initialized with a parameter named *contextConfigLocation* which contains the path to the configuration *XML*

load-on-startup is an integer value that specifies the order for multiple servlets to be loaded. So if you need to declare more than one servlet you can define in which order they will be initialized. Servlets marked with lower integers are loaded before servlets marked with higher integers.

Now our servlet is configured. The second step is declaring a *servlet-mapping*:

```
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

With the servlet mapping we are bounding it by its name to a *URL pattern* that specifies what *HTTP* requests will be handled by it.

4. Hybrid Configuration

With the adoption of the version 3.0 of *Servlet APIs*, the *web.xml* file has become optional, and we can now use Java to configure the *DispatcherServlet*.

We can register a servlet implementing a *WebApplicationInitializer*. This is the equivalent of the *XML* configuration above:

```
public class MyWebAppInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext container) {
        XmlWebApplicationContext context = new XmlWebApplicationContext();
        context.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

        ServletRegistration.Dynamic dispatcher = container
          .addServlet("dispatcher", new DispatcherServlet(context));

        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }
}
```

In this example we are:

1. Implementing the *WebApplicationInitializer* interface
2. Overriding the *onStartup* method we create a new *XmlWebApplicationContext* configured with the same file passed as *contextConfigLocation* to the servlet in the *XML* example

3. Then we are creating an instance of *DispatcherServlet* with the new context that we just instantiated
4. And finally we are registering the servlet with a mapping *URL pattern*

So we used *Java* to declare the servlet and bind it to a *URL mapping* but we kept the configuration in a separated *XML* file: *dispatcher-config.xml*.

5. 100% *Java* Configuration

With this approach our servlet is declared in Java, but we still need an *XML* file to configure it. With *WebApplicationInitializer* you can achieve a 100% *Java* configuration.

Let's see how we can refactor the previous example.

The first thing we will need to do is create the application context for the servlet.

This time we will use an annotation based context so that we can use *Java* and annotations for configuration and remove the need for *XML* files like *dispatcher-config.xml*:

```
AnnotationConfigWebApplicationContext context
  = new AnnotationConfigWebApplicationContext();
```

This type of context can then be configured registering a configuration class:

```
context.register(AppConfig.class);
```

Or setting an entire package that will be scanned for configuration classes:

```
context.setConfigLocation("com.example.app.config");
```

Now that our application context is created, we can add a listener to the *ServletContext* that will load the context:

```
container.addListener(new ContextLoaderListener(context));
```

The next step is creating and registering our dispatcher servlet:

```
ServletRegistration.Dynamic dispatcher = container
  .addServlet("dispatcher", new DispatcherServlet(context));

dispatcher.setLoadOnStartup(1);
dispatcher.addMapping("/");
```

Now our *WebApplicationInitializer* should look like this:

```
public class MyWebAppInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext container) {
        AnnotationConfigWebApplicationContext context
          = new AnnotationConfigWebApplicationContext();
        context.setConfigLocation("com.example.app.config");

        container.addListener(new ContextLoaderListener(context));

        ServletRegistration.Dynamic dispatcher = container
          .addServlet("dispatcher", new DispatcherServlet(context));

        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }
}
```

Java and annotation configuration offers many advantages. Usually it leads to shorter and more concise configuration and annotations provide more context to declarations, as it's co-located with the code that they configure.

But this is not always a preferable or even possible way. For example some developers may prefer keeping their code and configuration separated, or you may need to work with third party code that you can't modify.

6. Conclusion

In this article we covered different ways to configure a *DispatcherServlet* in *Spring 3.2+* and it's up to you to decide which one to use based on your preferences. *Spring* will accommodate to your decision whatever you choose.

You can find the source code from this article on Github here (https://github.com/eugenp/tutorials/tree/master/spring-mvc-java) and here (https://github.com/eugenp/tutorials/tree/master/spring-mvc-xml).

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)

REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)

JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)

SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)

SERIES

ABOUT

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)