https://dzone.com/articles/understanding-spring-web

**Understanding Spring Web Initialization**

A few years ago the majority of us were used to writing XML config files everywhere, to setup even a simple Java EE application. Today using Java or Groovy to configure projects is becoming preferred way - you just need to take a look at Gradle or functionalities introduced in further versions of the Spring Framework to gen up on this.  
  
Now I'll deal with configuring Spring contexts for web application.  
  
Java EE provides [ServletContainerInitializer](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletContainerInitializer.html) interface, which allows libraries to be notified of a web application startup. Since **Spring 3.1** we have [SpringServletContainerInitializer](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/SpringServletContainerInitializer.html) class which handles[WebApplicationInitializer](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/WebApplicationInitializer.html) by instantiating all found classes implementing this interface, sorting them basing on [@Order](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/annotation/Order.html) annotation (non-annotated classes gets the highest possible order, so they are processed at the end) and invoking *onStartup()* method.  
  
  
Spring since version **3.2** provides us a few classes implementing [WebApplicationInitializer](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/WebApplicationInitializer.html) interface, from which first is [AbstractContextLoaderInitializer](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/context/AbstractContextLoaderInitializer.html). This class included in spring-web module uses abstract *createRootApplicationContext()* method to create application context, delegates it to[ContextLoaderListener](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/context/ContextLoaderListener.html) which then is being registered in the [ServletContext](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletContext.html) instance. Creating application context using this class looks as follows:

public class SpringAnnotationWebInitializer extends AbstractContextLoaderInitializer {

@Override

protected WebApplicationContext createRootApplicationContext() {

AnnotationConfigWebApplicationContext applicationContext = new AnnotationConfigWebApplicationContext();

applicationContext.register(SpringAnnotationConfig.class);

return applicationContext;

}

}

That was the simplest way to start up Spring web context. But if we want to experience benefits provided by Spring MVC and don't want to manually register [DispatcherServlet](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html) it'll be better to use another class: [AbstractDispatcherServletInitializer](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/support/AbstractDispatcherServletInitializer.html). It extends previous class and adds two abstract methods: *createServletApplicationContext()* and *getServletMappings()*.  First method returns[WebApplicationContext](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/context/WebApplicationContext.html) that will be passed to [DispatcherServlet](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html), which will be automatically added into container [ServletContext](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletContext.html). Please notice that this context will be established as a child of the context returned by *createRootApplicationContext()* method. Second method - as you have probably already deduced - returns mappings that are used during servlet registration. You can also override *getServletFilters()* method if you need any custom filters, because default implementation returns just empty array. Exemplary implementation using this class could be:

public class SpringWebMvcInitializer extends AbstractDispatcherServletInitializer {

@Override

protected WebApplicationContext createRootApplicationContext() {

AnnotationConfigWebApplicationContext applicationContext = new AnnotationConfigWebApplicationContext();

applicationContext.register(SpringRootConfig.class);

return applicationContext;

}

@Override

protected WebApplicationContext createServletApplicationContext() {

AnnotationConfigWebApplicationContext applicationContext = new AnnotationConfigWebApplicationContext();

applicationContext.register(SpringMvcConfig.class);

return applicationContext;

}

@Override

protected String[] getServletMappings() {

return new String[]{"/\*"};

}

}

And now last but definitely not least class: [AbstractAnnotationConfigDispatcherServletInitializer](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/support/AbstractAnnotationConfigDispatcherServletInitializer.html). Here we can see further step in simplifying Spring initialization - we don't need to manually create contexts but just set appropriate config classes in *getRootConfigClasses()*and *getServletConfigClasses()* methods. I hope you are already familiar with those names, because they works exactly like in the former case. Of course due to this class extends [AbstractDispatcherServletInitializer](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/support/AbstractDispatcherServletInitializer.html) we can still override *getServletFilters()*. Finally we can implement our configuration in the following way:

public class SpringWebMvcSimpleInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected Class<?>[] getRootConfigClasses() {

return new Class[] {SpringRootConfig.class};

}

@Override

protected Class<?>[] getServletConfigClasses() {

return new Class[] {SpringMvcConfig.class};

}

@Override

protected String[] getServletMappings() {

return new String[]{"/\*"};

}

}

If you like to see wider context please follow examples in my GitHub repo: <https://github.com/jkubrynski/spring-java-config-samples/>