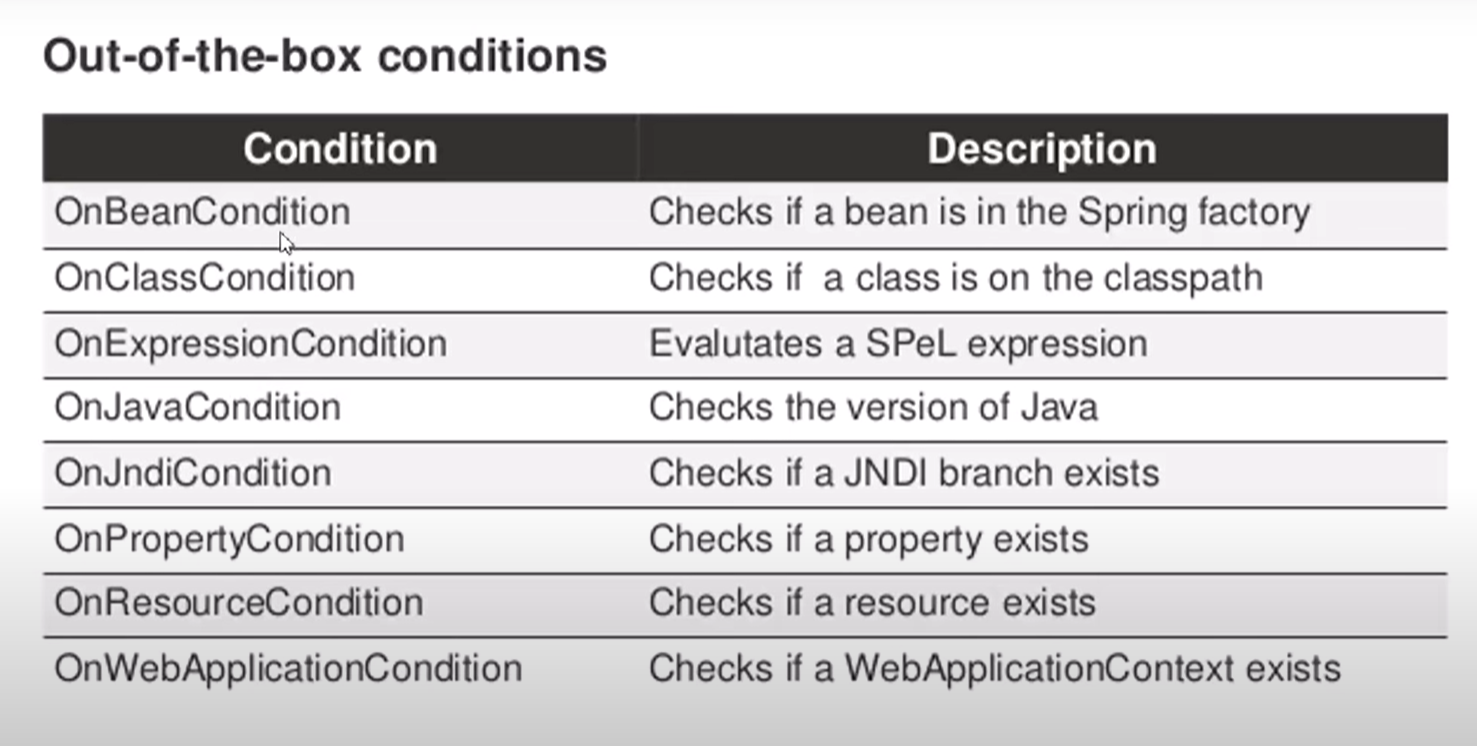
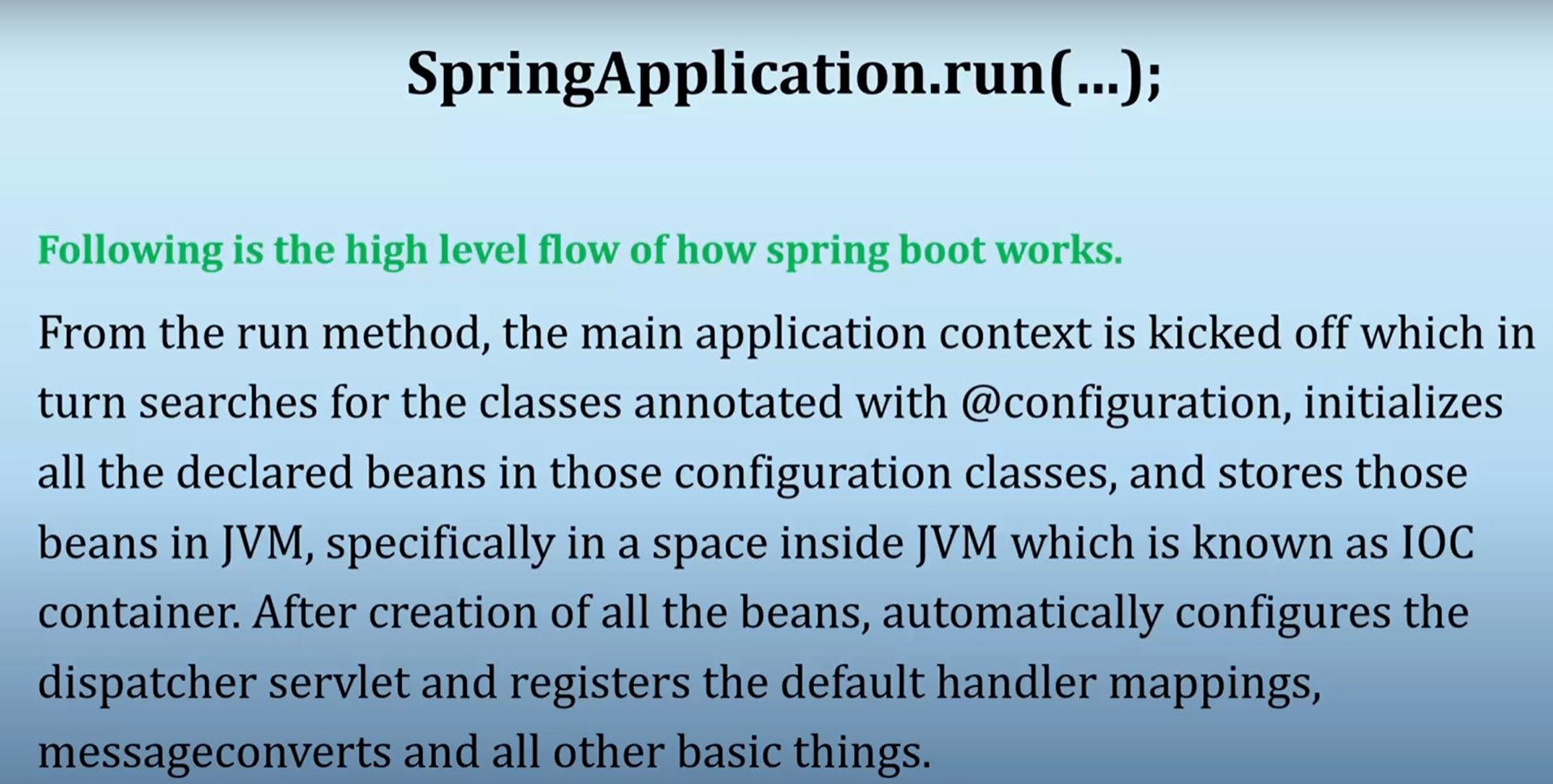


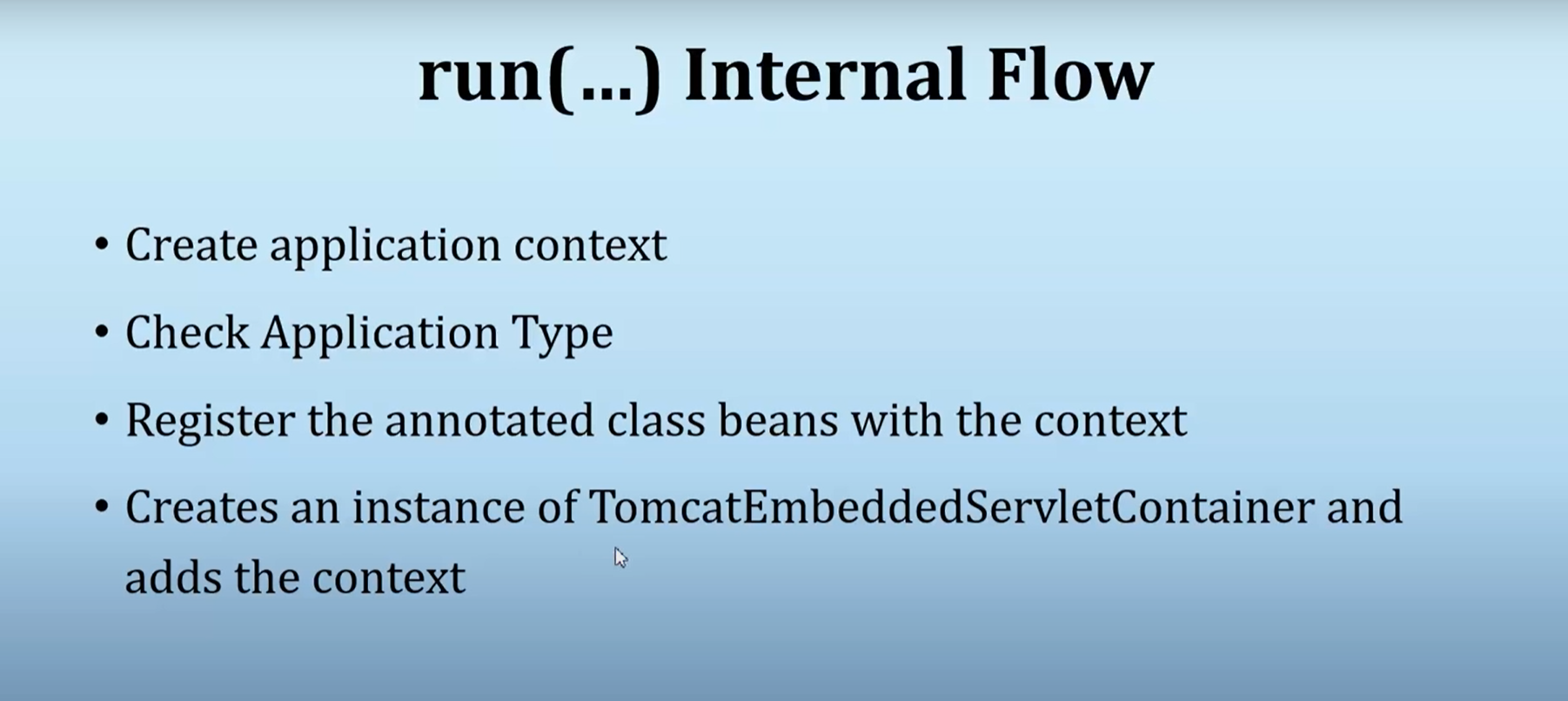
Mysql, web, spring data jpa

If you add all above dependency + dependencies added by spring boot you can see all the configuration in following path :

spring-boot-autoconfigure => META-INF => spring.factories







@SpringBootApplication source code

SpringApplication source code

Application type – applicationcontext/webapplicationcontext…/reactive

**Why do we need Spring Boot Auto Configuration?**

Spring based applications have a lot of configuration.

When we use Spring MVC, we need to configure component scan, dispatcher servlet, a view resolver, web jars(for delivering static content) among other things.

<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix">

<value>/WEB-INF/views/</value>

</property>

<property name="suffix">

<value>.jsp</value>

</property>

</bean>

<mvc:resources mapping="/webjars/\*\*" location="/webjars/"/>

Below code snippet shows typical configuration of a dispatcher servlet in a web application.

<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/todo-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

When we use Hibernate/JPA, we would need to configure a datasource, an entity manager factory, a transaction manager among a host of other things.

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"

destroy-method="close">

<property name="driverClass" value="${db.driver}" />

<property name="jdbcUrl" value="${db.url}" />

<property name="user" value="${db.username}" />

<property name="password" value="${db.password}" />

</bean>

<jdbc:initialize-database data-source="dataSource">

<jdbc:script location="classpath:config/schema.sql" />

<jdbc:script location="classpath:config/data.sql" />

</jdbc:initialize-database>

<bean

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"

id="entityManagerFactory">

<property name="persistenceUnitName" value="hsql\_pu" />

<property name="dataSource" ref="dataSource" />

</bean>

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

<property name="entityManagerFactory" ref="entityManagerFactory" />

<property name="dataSource" ref="dataSource" />

</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>

Above examples are typical with any Spring framework implementation or integration with other frameworks.

**Spring Boot : Can we think different?**

Spring Boot brings in new thought process around this.

*Can we bring more intelligence into this? When a spring mvc jar is added into an application, can we auto configure some beans automatically?*

* How about auto configuring a Data Source if Hibernate jar is on the classpath?
* How about auto configuring a Dispatcher Servlet if Spring MVC jar is on the classpath?

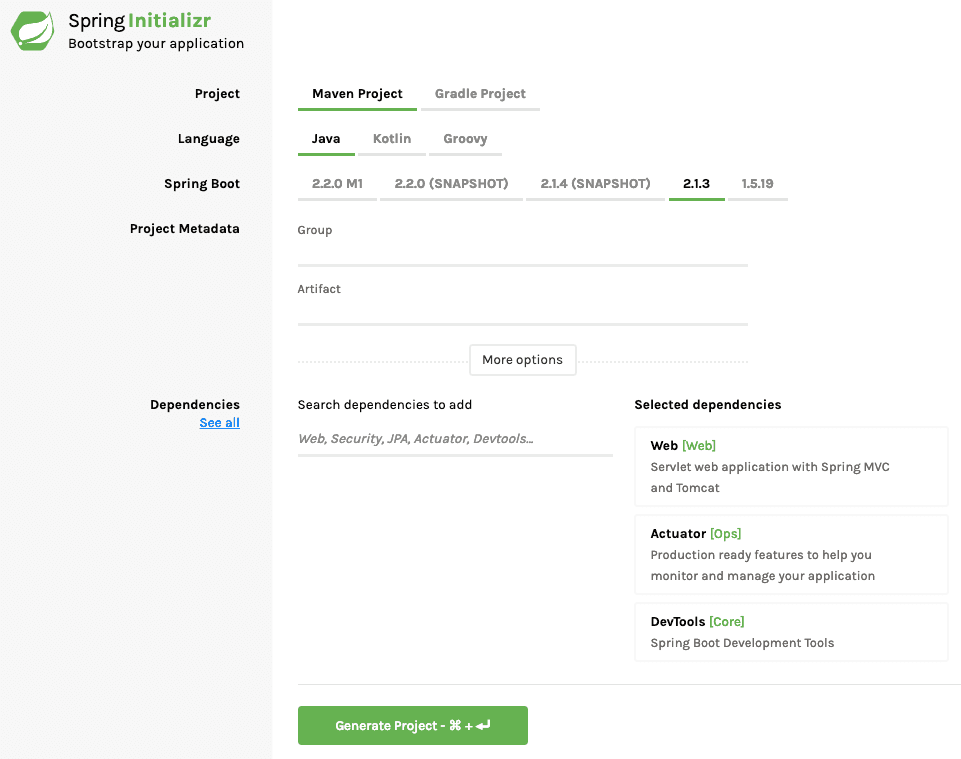
There would be provisions to override the default auto configuration.

*Spring Boot looks at a) Frameworks available on the CLASSPATH b) Existing configuration for the application. Based on these, Spring Boot provides basic configuration needed to configure the application with these frameworks. This is called Auto Configuration.*

To understand Auto Configuration further, lets bootstrap a simple Spring Boot Application using Spring Initializr.

**Creating REST Services Application with Spring Initializr**

*Spring Initializr*[*http://start.spring.io/*](https://start.spring.io/)*is great tool to bootstrap your Spring Boot projects.*



As shown in the image above, following steps have to be done.

* Launch Spring Initializr and choose the following
  + Choose com.in28minutes.springboot as Group
  + Choose student-services as Artifact
  + Choose following dependencies
    - Web
    - Actuator
    - DevTools
* Click Generate Project.
* Import the project into Eclipse.

**Spring Boot Auto Configuration in action.**

When we run StudentServicesApplication.java as a Java Application, you will see a few important things in the log.

Mapping servlet: 'dispatcherServlet' to [/]

Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequest)

Mapped URL path [/webjars/\*\*] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]

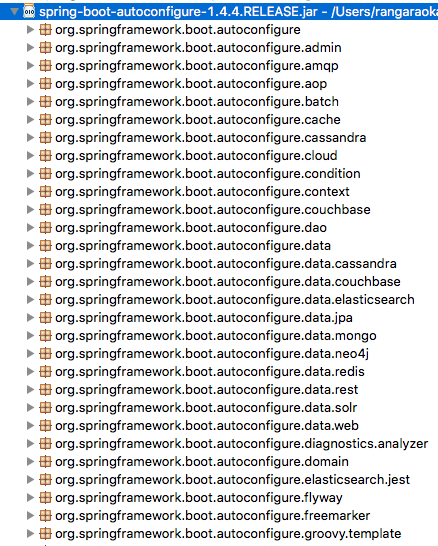
Above log statements are good examples of Spring Boot Auto Configuration in action.

As soon as we added in Spring Boot Starter Web as a dependency in our project, Spring Boot Autoconfiguration sees that Spring MVC is on the classpath. It autoconfigures dispatcherServlet, a default error page and webjars.

If you add Spring Boot Data JPA Starter, you will see that Spring Boot Auto Configuration auto configures a datasource and an Entity Manager.

**Where is Spring Boot Auto Configuration implemented?**

All auto configuration logic is implemented in spring-boot-autoconfigure.jar. All auto configuration logic for mvc, data, jms and other frameworks is present in a single jar.



Other important file inside spring-boot-autoconfigure.jar is /META-INF/spring.factories. This file lists all the auto configuration classes that should be enabled under the EnableAutoConfiguration key. A few of the important auto configurations are listed below.

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\

org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\

org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\

org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\

org.springframework.boot.autoconfigure.security.SecurityFilterAutoConfiguration,\

org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\

org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\

org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration,\

**Example Auto Configuration**

We will take a look at DataSourceAutoConfiguration.

Typically all Auto Configuration classes look at other classes available in the classpath. If specific classes are available in the classpath, then configuration for that functionality is enabled through auto configuration. Annotations like @ConditionalOnClass, @ConditionalOnMissingBean help in providing these features!

@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class }) : This configuration is enabled only when these classes are available in the classpath.

@Configuration

@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })

@EnableConfigurationProperties(DataSourceProperties.class)

@Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class })

public class DataSourceAutoConfiguration {

@ConditionalOnMissingBean : This bean is configured only if there is no other bean configured with the same name.

@Bean

@ConditionalOnMissingBean

public DataSourceInitializer dataSourceInitializer() {

return new DataSourceInitializer();

}

Embedded Database is configured only if there are no beans of type DataSource.class or XADataSource.class already configured.

@Conditional(EmbeddedDatabaseCondition.class)

@ConditionalOnMissingBean({ DataSource.class, XADataSource.class })

@Import(EmbeddedDataSourceConfiguration.class)

protected static class EmbeddedDatabaseConfiguration {

}

**Debugging Auto Configuration**

There are two ways you can debug and find more information about auto configuration.

* Turning on debug logging
* Using Spring Boot Actuator

**Debug Logging**

You can turn debug logging by adding a simple property value to application.properties. In the example below, we are turning on Debug level for all logging from org.springframework package (and sub packages).

logging.level.org.springframework: DEBUG

When you restart the application, you would see an auto configuration report printed in the log. Similar to what you see below, a report is produced including all the auto configuration classes. The report separates the positive matches from negative matches. It will show why a specific bean is auto configured and also why something is not auto configured.

=========================

AUTO-CONFIGURATION REPORT

=========================

Positive matches:

-----------------

DispatcherServletAutoConfiguration matched

- @ConditionalOnClass classes found: org.springframework.web.servlet.DispatcherServlet (OnClassCondition)

- found web application StandardServletEnvironment (OnWebApplicationCondition)

Negative matches:

-----------------

ActiveMQAutoConfiguration did not match

- required @ConditionalOnClass classes not found: javax.jms.ConnectionFactory,org.apache.activemq.ActiveMQConnectionFactory (OnClassCondition)

AopAutoConfiguration.CglibAutoProxyConfiguration did not match

- @ConditionalOnProperty missing required properties spring.aop.proxy-target-class (OnPropertyCondition)