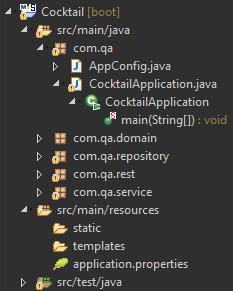
Spring Properties and Configuration

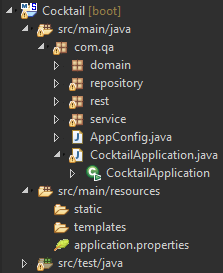
# Project Structure

An important point to highlight early on is the package structure of our projects. The class that has our main method in it, conventionally called Application or some variation, needs to be at the root of the rest of your Java classes. This is very important for certain annotations to work correctly.



In this example, CocktailApplication.java is in the package *com.qa,* other java code will be in packages **within** this package, for instance *com.qa.domain.*

You can switch your Package Explorer to a **hierarchical** view in STS that I think better represents this.



# application.properties

The default location for Spring configuration, application.properties allows for the setting of a massive amount of Spring properties. A list of common properties can be found at <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

This file is empty by default, why? Because Spring Boot is opinionated and does the initial configuration for you.

# Beans.xml

In Spring it is possible to configure beans using both Java and XML configuration. While the Java configuration is a lot newer it is still worth knowing how to use the XML method as it is still an entirely valid of creating your beans and will still be in use in many legacy systems.

By convention the file where you set up your beans should be called Beans.xml and can be imported into your application using the **@ImportResource** annotation on an otherwise empty class, like so:

Configuration

@ImportResource("<Path to xml file>")

**public class BeansConfig {**

// empty class

**}**

Inside Beans.xml beans are defined in this format:

<bean id=*"hello"* class=*"com.qa.demo.helloWorld"* scope=*"singleton"*></bean>

This initialises the name of the bean, which class to create it from and what scope should be applied to the bean after it has been created.

It is possible to **wire** your beans together using XML configuration; for example:

<bean id=*"message"* class=*"com.qa.demo.Message"* scope=*"prototype"*></bean>

<bean id=*"messager"* class=*"com.qa.demo.Messager"* scope=*"singleton"*>

<property name=*"messageField"* ref=*"message"*></property>

</bean>

Using this configuration, a bean called “**message**” will be created from the **Message** class and injected into the “**messager**” bean. This essentially sets the value of the “**message**” field in the **Messager** class equal to the “**message**” bean.

This example is available at: <https://gist.github.com/JHarry444/96e9377bb4066663c369e9fbb4e6547d>

# @Component

This is an **extremely** useful annotation when working with Spring Boot as it enables a class to be picked up by **component scanning** (which is enabled by **@ComponentScan**, which is one of the annotations that make up **@SpringBootApplication**) and converted into beans automatically, saving the developer a lot of time and boiler plate code.

At this point it is really important to revisit the project structure of our Spring Boot application, else the **@ComponentScan** will not work correctly.

Broadly speaking, components can be broken up into four categories:

## @Repository

Repositories represent **Data** **Access** **Objects**, i.e. they are how Spring fetches objects from databases. Using a repository gives you a unified way to access any kind of database and allows any exceptions thrown to be converted to Spring’s own **DataAccessException**, significantly simplifying error handling when accessing databases.

## @Service

Has no specific functionally as of yet but conventionally it is used to signify a class contains the **business logic** for the application. The Spring documentation states that this annotation may have additional functionality added to it in the future so it’s definitely best practice to use it but at the moment it functions more as a **statement of intent** than anything else.

## @Controller

Methods in controller classes can have annotations like **@RequestMapping** applied to them in order to expose them as end points. This allows access to the application via HTTPRequests. **@Controller** has its own subtype called **@RestController** that specifies that its methods will return objects in the form of JSON, which would otherwise be accomplished using the **@ResponseBody** annotation.

# Miscellaneous Notes

When you retrieve a bean we use the following syntax – ApplicationContext.getBean(<name of bean>), remember to pass in the name of the bean you want to access and not the method of the bean.

So in the example in the slides the **MyComponent** class that you are attaching the @Component too will be referred to as **myComponent** by the IoC. Therefore when we want to get the bean we have to use ApplicationContext.getBean(**myComponent**).

This casting from Pascal case to Camel Case is done automatically.

## 