

Frameworks and Libraries

Oliver Drotbohm – 2020-10-25 13:46:34 +0100

| This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

◀ Back to lectures

1. Introduction

1.1. Frameworks VS. Libraries

2. JUnit — a test framework

2.1. Unit- VS. Integration tests

3. A Java web application — Quick start

3.1. Objectives

3.2. Create a new project

3.3. Add minimal functionality and run this thing

3.4. Make this a bit more dynamic

4. Spring — An application framework

4.1. Inversion of Control

4.2. Fundamental building blocks of a Spring Boot application

4.3. Running a Spring Boot application

4.3.1. Standalone

4.3.2. In integration tests

4.4. Application components

5. Sample class design

Appendix

Appendix A: License

1. Introduction

This script is supposed to give an overview about the concept of a library and a framework and distinguish one from another. We're going to look at JUnit as an example of a test framework and Spring as an example of a general purpose application framework.

1.1. Frameworks VS. Libraries

- Library: your code uses the provided code (e.g. Google Guava, Spring JDBC)
 - Framework: the provided code uses your code (e.g. JUnit, Spring Framework)
-

2. JUnit — a test framework

JUnit is a Java based test framework that — contrary to what its name implies — can be used to write both unit tests and integration tests. Until version 3, it used an inheritance based approach for its user-facing API. That has unanimously been considered too invasive so that the current framework generation relies on annotations.

2.1. Unit- VS. Integration tests

Software can be tested on multiple levels of granularity: individual components can be tested in isolation, in the context of collaborating components. The entire system can be tested as a whole.

The most fundamental approaches to testing are called unit tests. They rely on the component to be testable with a minimal amount of context available. It's usually desirable to use a class on its own, potentially replacing required collaborators with test mocks or stubs. To make sure the effort for that is manageable, this requires the class under test to be used in an isolated way.

// The four primary reasons developers write tests:

1. Prevent regressions
2. Improve design
3. Enable later refactoring
4. Document behavior

— Sarah Mei - @sarahmei

Thus writing unit tests for a component usually drives the design of that class to become more simple. The approach of writing test cases to drive the design of code is called Test Driven Design/Development (TDD^[1])



3. A Java web application — Quick start

3.1. Objectives

- Learn how to create, setup and run a Java based web project
- Learn how to handle a web request
- Learn how to efficiently work with web related code

3.2. Create a new project

IDE centric style

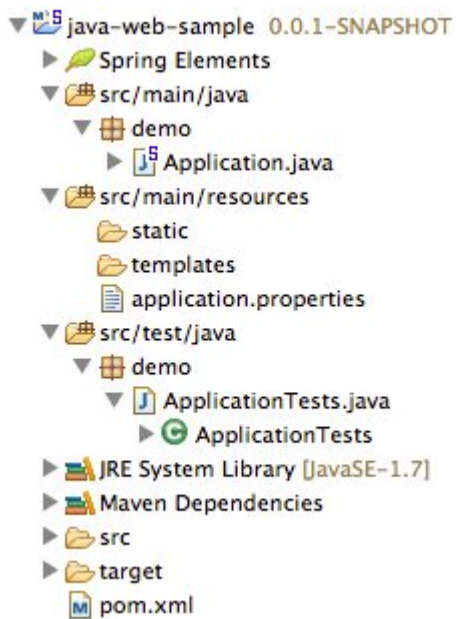
1. Start STS.
2. Create a new project **File > New... > Other** (or  + ).
3. Select "Spring Starter Project".
4. Keep settings (details on that below).
5. Select "Web" in the Dependencies matrix.
6. Select [**Finish**].

Manual version

1. Browse to <http://start.spring.io>.
2. Keep settings (details on them below).
3. Select "Web" in the Dependencies matrix.
4. Select [**Generate project**].
5. Open up your IDE.
6. In Eclipse select **File > Import....**
7. Select "Existing Maven Projects".

The result

The result should be a Java project created in the IDE looking something like this:



What do we have here?

- `Application.java` in `src/main/java` - The main (startable) application class.
- `static` / `templates` folders in `src/main/resources` - Folders for static resources and view templates.
- `application.properties` - Configuration file.
- `ApplicationTests.java` - Integration test to test the application.
- `pom.xml` - The POM (Project Object Model). A Maven configuration file to control project setup, build and dependencies.

3.3. Add minimal functionality and run this thing

- Create a new class, e.g. in the `demo` package: Select `src/main/java`, press `⌘` + `⌘` + `N`, type `Class` or select **File** > **New...** > **Class** to bring up the new class wizard. Enter `demo` for the package name, `WelcomeController` as type name. Edit the created class to look like this:

Example 1. A simple Spring MVC controller

```
@RestController 1
class WelcomeController {
```

JAVA

```

@RequestMapping("/welcome") 2
String welcome() {
    return "Welcome"; 3
}
}

```

- Causes the class to be found by the Spring container and turned into a Spring bean. This is not directly caused by `@RestController` but the annotation
- 1 being meta-annotated with `@Controller` which is in turn annotated with `@Component`. See more on that in [Fundamental building blocks of a Spring Boot application](#).
 - 2 Maps `/welcome` to the execution of the annotated method. All HTTP request to that URI will cause the method to be invoked.
 - 3 The result to be written to the response. Using `@RestController` causes the `String` returned to be considered the response payload.

When pasting this code into your IDE it might already import the right types for you. In case it doesn't and you see red underlines e.g. under `@RestController` you have a variety of options:

- Navigate to right after `@RestController` and hit `^` + `Space`. This will trigger code completion suggestions and either provide you with a drop down list to choose the type from (in case multiple ones matching what you typed are on the classpath).
 - Alternatively try `⌘` + `⬆` + `⌘` (Organize imports), which should try to resolve the type tokens you used.
- Execute `Application.java` (`⌘` + `⌘` + `x`, `j` or right click the class, **Run As > Java Application**).
 - Browse to <http://localhost:8080/welcome>, should give you `Welcome`.




Try using `⌘` + `Space` while typing here and there. Eclipse will usually provide you quite useful suggestions for code completion.

3.4. Make this a bit more dynamic

If you run the application in debug mode, you can change method bodies on the fly without having to restart the app. We're going to use that mode going forward.

- Change the method signature of the controller method to this:

```
welcome(@RequestParam(value = "name") Optional<String> name)
```

- Restart the application in debug mode ( +  + , **J** or **Debug As > Java Application**).
- Browse to <http://localhost:8080/welcome>, output should be unchanged.
- Change the URI to <http://localhost:8080/welcome?name=Java>, doesn't have any effect yet.
- While the application is running, change the method body to:

```
return "Welcome ".concat(name.orElse("World")).concat("!");
```

- Refresh the browser tab and the response should now adapt to changes of the `name` attribute.

4. Spring — An application framework

The Spring Framework ^[2] is an open-source, general purpose, Java-based application framework. Parts of it can be used as a library but at the very core of it it's an inversion-of-control^[3] container which allows you to write easy to test application components.

Beyond that it provides means to apply technical services (e.g. transactions and security) in a declarative and portable way, integrations with persistence and integration technologies as well as an MVC web framework.

As a developer you usually use Spring Framework through Spring Boot, which is an opinionated add-on to automatically configure and ease the set up of a Spring based application so that you can focus on your business code^[4].

4.1. Inversion of Control

Inversion of Control (otherwise also referred to as Dependency Injection) is the software design technique that frees a component from the task to look up its collaborators and rather get those handed into the component – usually as constructor arguments. This inverts the control of this lookup from the component itself to its context (hence the name). The assembly of a component is externalized and usually taken care of by a generic framework but can — and almost always is — also be used manually e.g. in test cases.

This approach especially improves testability of components as the collaborators that would be used in a production environment can easily be replaced with test components.

4.2. Fundamental building blocks of a Spring Boot application

- **Application code** — code you write, e.g. the `WelcomeController` in the quick start example.
- **Configuration** — code to configure the application container. Declares references to infrastructure components (e.g. the database, security) and defines how application components are found:

Example 2. An example configuration class

```

@SpringBootApplication
class Application {

    public static void main(String... args) {
        SpringApplication.run(Application.class, args);
    }
}

```

This is the primary application class (hence the name) and uses `SpringApplication.run(...)` in the main method to start a Spring `ApplicationContext` and inspect the given configuration, the `Application` class itself in this case.

The `@SpringBootApplication` is the annotation to enable very fundamental features of Spring boot. It actually acts like a combination of the following three annotations:

- `@Configuration` — Declares the class to contain configuration. This will cause the container to detect methods annotated with `@Bean` for you to provide custom components manually.
- `@EnableAutoConfiguration` — Enables Spring Boot's auto-configuration features to activate features based on your classpath setup.
- `@ComponentScan` — Activates the detection of Spring components for the package of the annotated class.

Thus, using the single annotation, all of these features get activated at application startup.

4.3. Running a Spring Boot application

The application containing a main method, makes it very easy to start it from within the IDE as the class itself becomes executable. That's very convenient during development. However, there are a couple of more ways we might want to execute the application: packaged into an executable, so that we can actually run it on a production server, and during integration test execution to verify its behavior.

4.3.1. Standalone

- On the command line, run `mvn clean package` and run the JAR (Java application ARchive) using `java -jar target/*.jar`. You can basically take the JAR created by the build and run that on any machine that has Java installed.

Don't be trapped by trying to use your IDE's JAR export mechanism. It's crucial for the application to work as a standalone JAR to be built using the the actual build mechanism, which applies the necessary plugins to make the JAR bootable.

4.3.2. In integration tests

- One of the most common ways of executing Test cases is with an open-source library called `JUnit` which has both Maven and Eclipse integration.
- To bootstrap the application container in an integration test the test class has to look as follows:

Example 3. Bootstrapping the Spring container from an integration test


```

@SpringBootTest
@RunWith(SpringRunner.class)
class ApplicationTests { ... }

```

JAVA

- `@SpringBootTest` expresses Boot to be used during tests. The main configuration class will be auto-detected looking for a class with an `@SpringBootApplication` in the current package and all above.
- `@RunWith(...)` tells JUnit to give Spring Framework the control over the test execution.

4.4. Application components

Application components are usually identified by an annotation that is either `@Component` or an annotation which is annotated with `@Component` in turn (e.g. `@Service`, `@Repository`, `@Controller`). The component classes are discovered at bootstrap time and a single instance is created.

+ .A simple application component

```

@Component
class MyApplicationComponent {}

```

JAVA

If a component needs other components to work with (e.g. the web controller needs access to the component implementing data access), the component required can be injected into the depending component by declaring a constructor taking a dependency of the necessary type.

Example 4. A simple component with a dependency

```

@Component
class MyDependingComponent {

    private final MyApplicationComponent dependency;

    public MyDependingComponent(MyApplicationComponent dependency) {
        this.dependency = dependency;
    }
}

```

JAVA

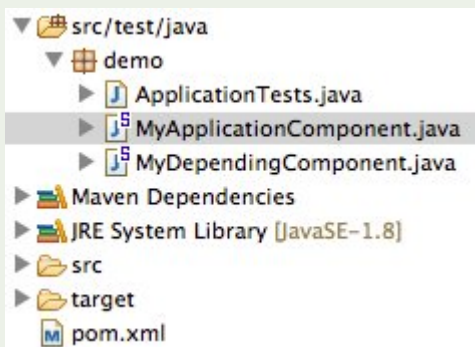
```
}  
}
```

If a component depended on cannot be found in the container, an exception is thrown:

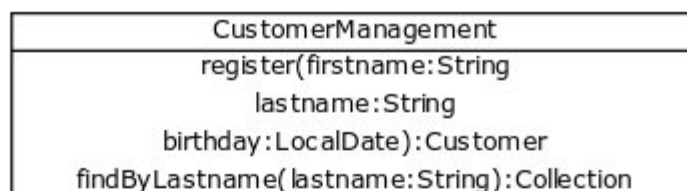
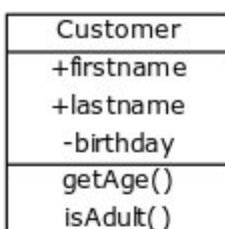
Example 5. A Spring exception indicating a component cannot be found

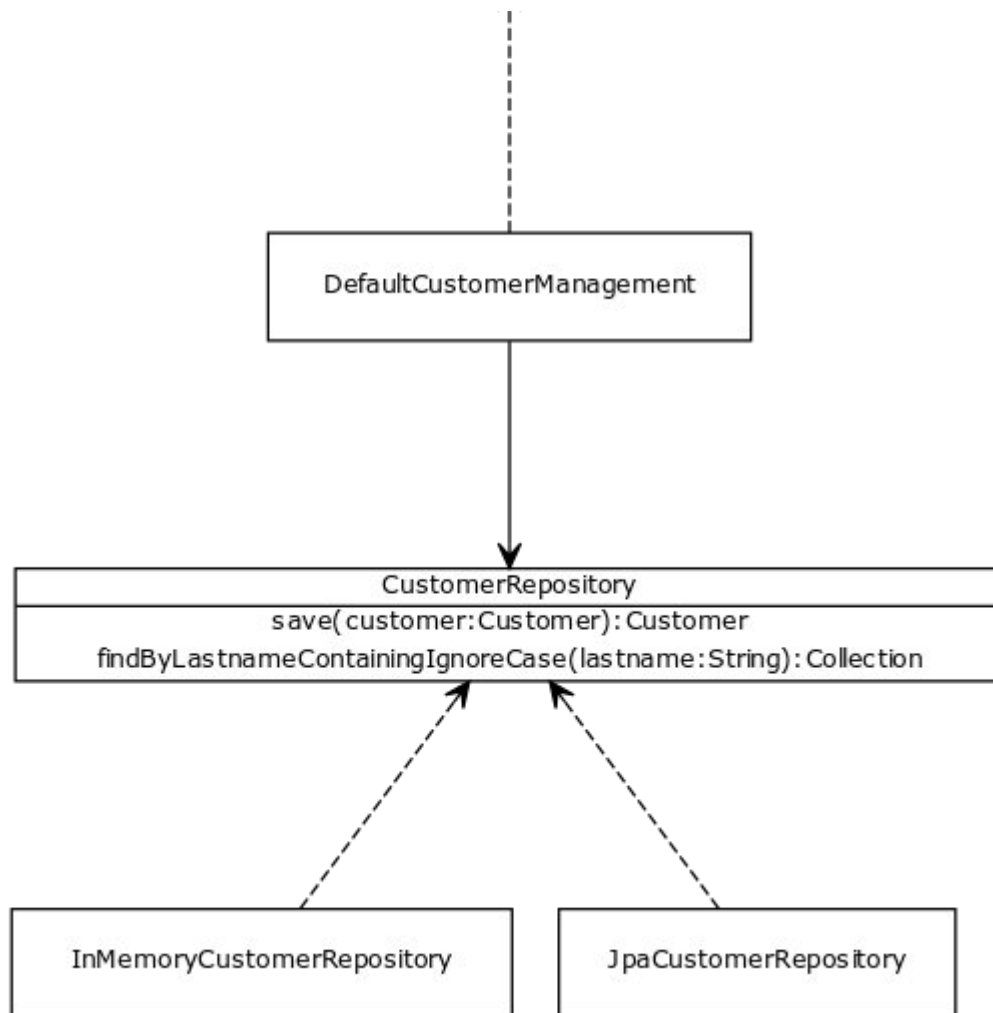
```
Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying  
    at o.s.b.f.s.DefaultListableBeanFactory.raiseNoSuchBeanDefinitionException(...:1118)  
    at o.s.b.f.s.DefaultListableBeanFactory.doResolveDependency(...:967)  
    at o.s.b.f.s.DefaultListableBeanFactory.resolveDependency(...:862)  
    at o.s.b.f.s.ConstructorResolver.resolveAutowiredArgument(...:811)  
    at o.s.b.f.s.ConstructorResolver.createArgumentArray(...:739)  
    ... 42 common frames omitted
```

When using STS, classes that are Spring components carry a little S-overlay on the icon:



5. Sample class design





Appendix

Appendix A: License



-
1. Test-Driven Development - [Wikipedia](#)
 2. Spring - [Wikipedia](#)
 3. Inversion of Control - [Wikipedia](#)
 4. Philosophies that Shaped Successful Frameworks - [Blog post](#)
-

Last updated 2020-10-25 13:46:34 +0100