Spring

Backend ITMO 2021

Spring Ecosystem

- 20+ active projects
  - Spring Framework
  - Spring Boot
  - Spring Data
  - Spring Cloud
  - ...
- spring.io
- All based on Spring Framework

```
@SpringBootApplication
@RestController
public class DemoApplication {

  @GetMapping("/helloworld")
  public String hello() {
    return "Hello World!";
  }
}
```

Spring Framework

- The most popular application development framework for enterprise Java
- Open source Java platform since 2003

- Dependency Injection (IoC)
- AOP
- Data Access / Integration
- Web
- Test

Dependency Injection / Inversion of Control

- **Dependency Injection**

  An injection is the passing of a dependency (a service) to a dependent object (a client). Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

- **Inversion of Control**

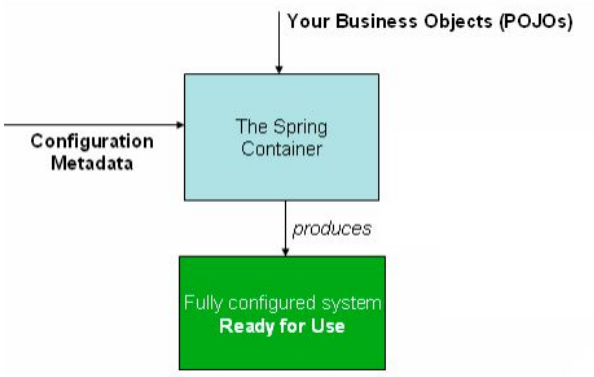
  In software engineering, inversion of control (IoC) describes a design in which custom-written portions of a computer program receive the flow of control from a generic, reusable library.

DI Sample

- Dependency
  - Class A depends on Class B
- DI realization
  - Inject by constructor
  - Inject by setter

IoC Container

- IoC Container is at the core of the Spring Framework
- The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction
- The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided (XML, Java Config)



* *POJO - plain old Java object*

DI Spring Sample

- Spring Beans - the objects that form the backbone of the application and that are managed by IoC Container
- Application Context - holds the information of all the beans that it instantiates

```java
@Configuration
public class TestConfiguration {
    @Bean
    Foo getFoo() {
        return new Foo();
    }

    @Bean
    Bar getBar() {
        return new Bar(getFoo());
    }
}
```

```java
public class Runner {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(TestConfiguration.class);
        Bar bar = context.getBean(Bar.class);
        System.out.println(bar);
    }
}
```

@Component, @ComponentScan

- @Component makes Spring Bean from Bar
- @ComponentScan scans package for components

```java
@Configuration
@ComponentScan
public class TestConfiguration {
    @Bean
    Foo getFoo() {
        return new Foo();
    }
}
```

```java
@Component
public class Bar {
    Foo foo;

    public Bar(Foo foo) {
        this.foo = foo;
    }
}
```

@Autowired

- Constructor Injection (annotation not required)
- Setter Injection
- Field Injection

```
@Configuration
@ComponentScan
public class TestConfiguration {
}
```

```
@Component
public class Foo {

}
```

```
@Component
public class Bar {
    Foo foo;

    public Bar(@Autowired Foo foo) {
        this.foo = foo;
    }
}
```

```
@Component
public class Bar {
    Foo foo;

    @Autowired
    public void setFoo(Foo foo) {
        this.foo = foo;
    }
}
```

```
@Component
public class Bar {
    @Autowired Foo foo;
}
```

Injecting Collections

- Interface Foo with many @Component impls
  - @Autowired List<Foo> foos;
  - @Autowired Foo[] foos;
- Ordering components
  - @Order(value = 1)
- Empty collection as default value
  - @Autowired(required = false)
  - List<Foo> foos = new ArrayList<>();
- Filtering by Qualifier
  - Set the same annotation @Qualifier(“qualifier_name”) for component and autowired field

Bean Scopes

- singleton (default)
- prototype
- request
- session
- application
- websocket

- Bean with the **singleton** scope: the container creates a single instance of that bean; all requests for that bean name will return the same object, which is cached. Any modifications to the object will be reflected in all references to the bean.
- A bean with the **prototype** scope will return a different instance every time it is requested from the container.

Look at Code #1

Cars example

XML Configuration [1]

```
public class Country {
    private String name;
    private long population;

    public Country(String name, long population) {
        super();
        this.name = name;
        this.population = population;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public long getPopulation() {
        return population;
    }
    public void setPopulation(long population) {
        this.population = population;
    }

}
```

XML Configuration [2]

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
    ">

    <bean id="country" class="org.arpit.java2blog.model.Country">
        <constructor-arg index="0" value="India"></constructor-arg>
        <constructor-arg index="1" value="20000"></constructor-arg>
    </bean>
    <context:annotation-config />
</beans>
```

XML Configuration [3]

```
public class SpringXMLConfigurationMain
{
    public static void main( String[] args )
    {
        ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("ApplicationContext.xml");
        Country countryBean = (Country) ac.getBean("country");
        String name = countryBean.getName();
        System.out.println("Country name: "+name);
        long population = countryBean.getPopulation();
        System.out.println("Country population: "+population);
        ac.close();
    }
}
```

Maven

- **Maven**(Apache) is an open-source software **project management tool** that is primarily used for Java projects. It can also be used for other programming projects such as C#, Ruby, Scala, and more. Maven addresses the two main aspects of software development: dependency, and how software is built. In Maven, an XML file describes the building process of a project, its dependencies, components, and other external modules. There are predefined targets for tasks like packaging and compiling.

Gradle

- **Gradle** is a **build automation tool** that is an open-source and builds based on the concepts of Apache Maven and Apache Ant. It is capable of building almost any type of software. It is designed for the multi-project build, which can be quite large. It introduces a **Java/Groovy-based DSL** (Domain Specific Language) instead of XML (Extensible Markup Language) for declaring the project configuration. It uses a **DAG** (Directed Acyclic Graph) to define the order of executing the task. Gradle offers an elastic model that can help the development lifecycle from compiling and packaging code for web and mobile applications.

Gradle / Maven

Gradle	Maven
It is a build automation system that uses a Groovy-based DSL (domain-specific language)	It is a software project management system that is primarily used for java projects.
It does not use an XML file for declaring the project configuration.	It uses an XML file for declaring the project, its dependencies, the build order, and its required plugin.
It is based on a graph of task dependencies that do the work.	It is based on the phases of the fixed and linear model.
In Gradle, the main goal is to add functionality to the project.	In maven, the main goal is related to the project phase.
It avoids the work by tracking input and output tasks and only runs the tasks that have been changed. Therefore it gives a faster performance.	It does not use the build cache; thus, its build time is slower than Gradle.
Gradle is highly customizable; it provides a wide range of IDE support custom builds.	Maven has a limited number of parameters and requirements, so customization is a bit complicated.
Gradle avoids the compilation of Java.	The compilation is mandatory in Maven.

The POM

- The **pom.xml** file is the core of a project's configuration in Maven. It is a single configuration file that contains the majority of information required to build a project in just the way you want.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Look at Code #2

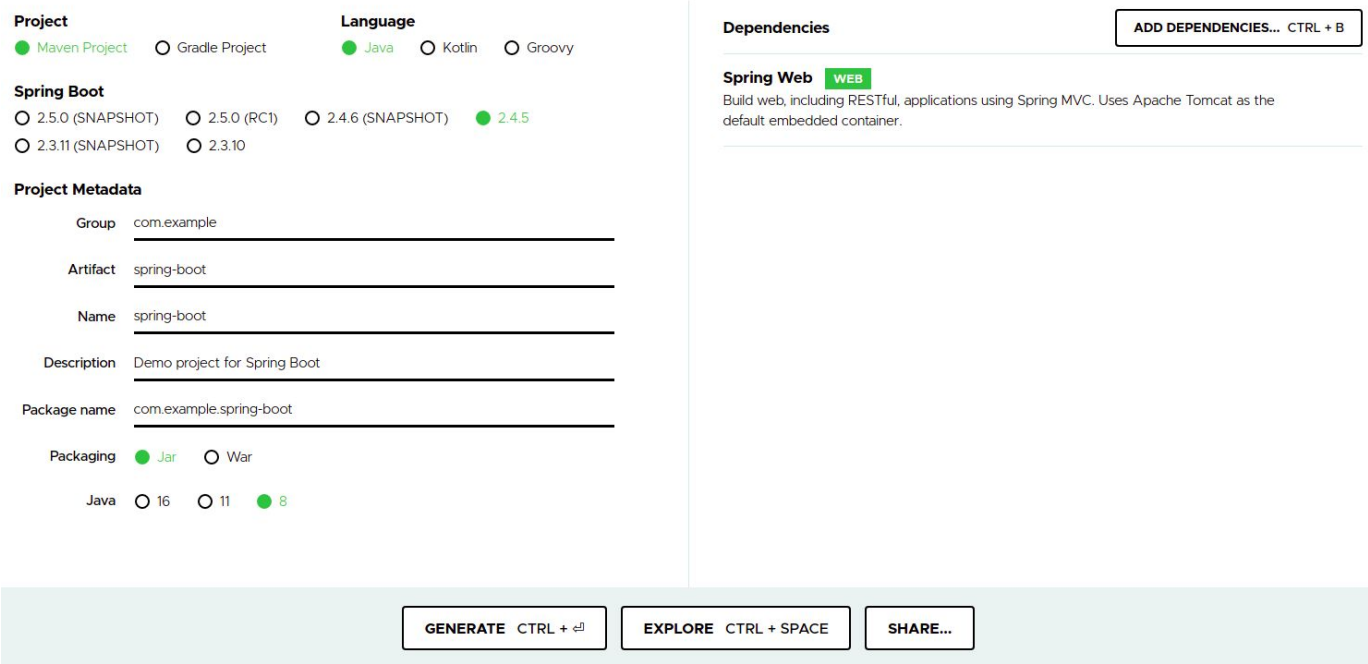
- Small MVN project example

Spring Boot

- Spring Boot offers a fast way to build applications. It looks at your classpath and at the beans you have configured, makes reasonable assumptions about what you are missing, and adds those items. With Spring Boot, you can focus more on business features and less on infrastructure.
- Spring Boot does not generate code or make edits to your files. Instead, when you start your application, Spring Boot dynamically wires up beans and settings and applies them to your application context.

Spring Initializr

spring initializr https://start.spring.io/



@SpringBootApplication

- Convenience annotation that adds all of the following:
  - **@Configuration**: Tags the class as a source of bean definitions for the application context.
  - **@EnableAutoConfiguration**: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.
  - **@ComponentScan**: Tells Spring to look for other components, configurations, and services in the com/example package, letting it find the controllers.

```
@SpringBootApplication
public class Application {

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

Spring Test

- The **@SpringBootTest** annotation tells Spring Boot to look for a main configuration class (one with **@SpringBootApplication**, for instance) and use that to start a Spring application context.
- The application context is cached between tests
- Uses JUnit

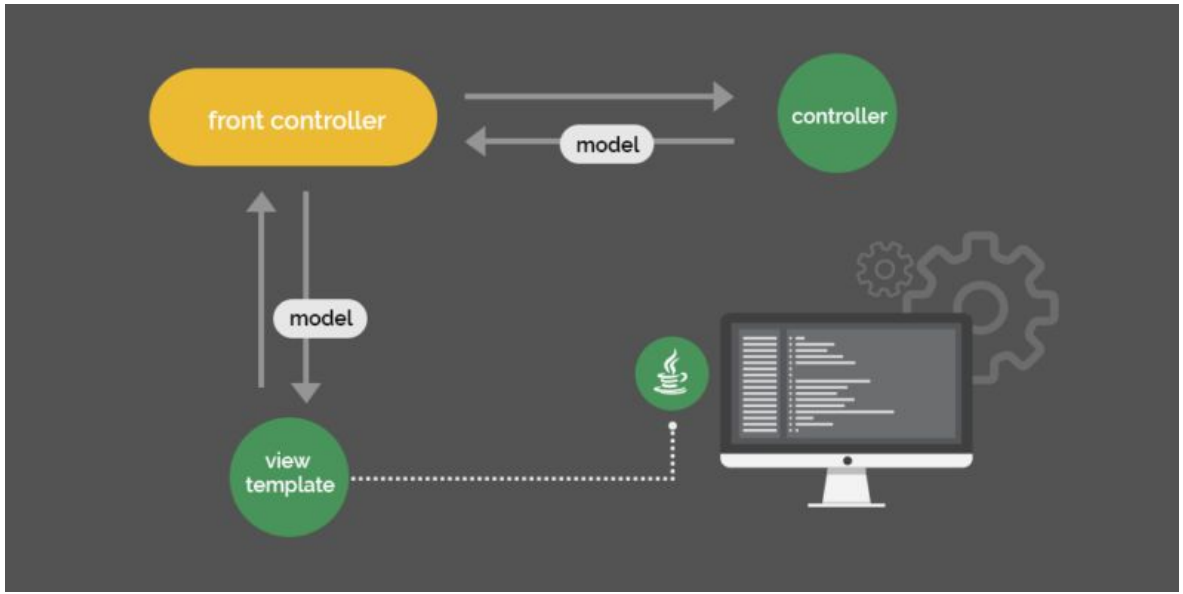
```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
</dependency>
```

Web Testing

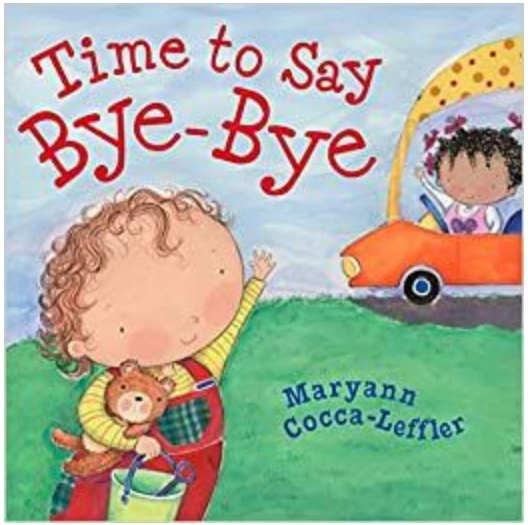
- Start app on random port, make requests
  - @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
  - Provided by **TestRestTemplate**
- Do not start the server at all but to test only the layer below that, where Spring handles the incoming HTTP request and hands it off to your controller. That way, almost of the full stack is used, and your code will be called in exactly the same way as if it were processing a real HTTP request but without the cost of starting the server.
  - Provided by **MockMvc** (@AutoConfigureMockMvc)
  - The full Spring application context is started but without the server

Look at Code #3

Creating app using Spring Boot and implementing sample endpoint



The End



SB

Backend ITMO 2021