Introduction to Spring Core and Spring Boot

Session: Spring Core
Instructor: Vishwa Mohan

Spring Core

Creating a Spring Project

MAVEN

- For developing big enterprise applications, you may need hundreds of jars.
- Downloading these hundreds of jars from different websites and adding them to the classpath can be quite tedious.
- This is where Maven comes into play.
- You only need to provide a list of dependencies needed for your application to the Maven, and it will download and add them to the classpath.

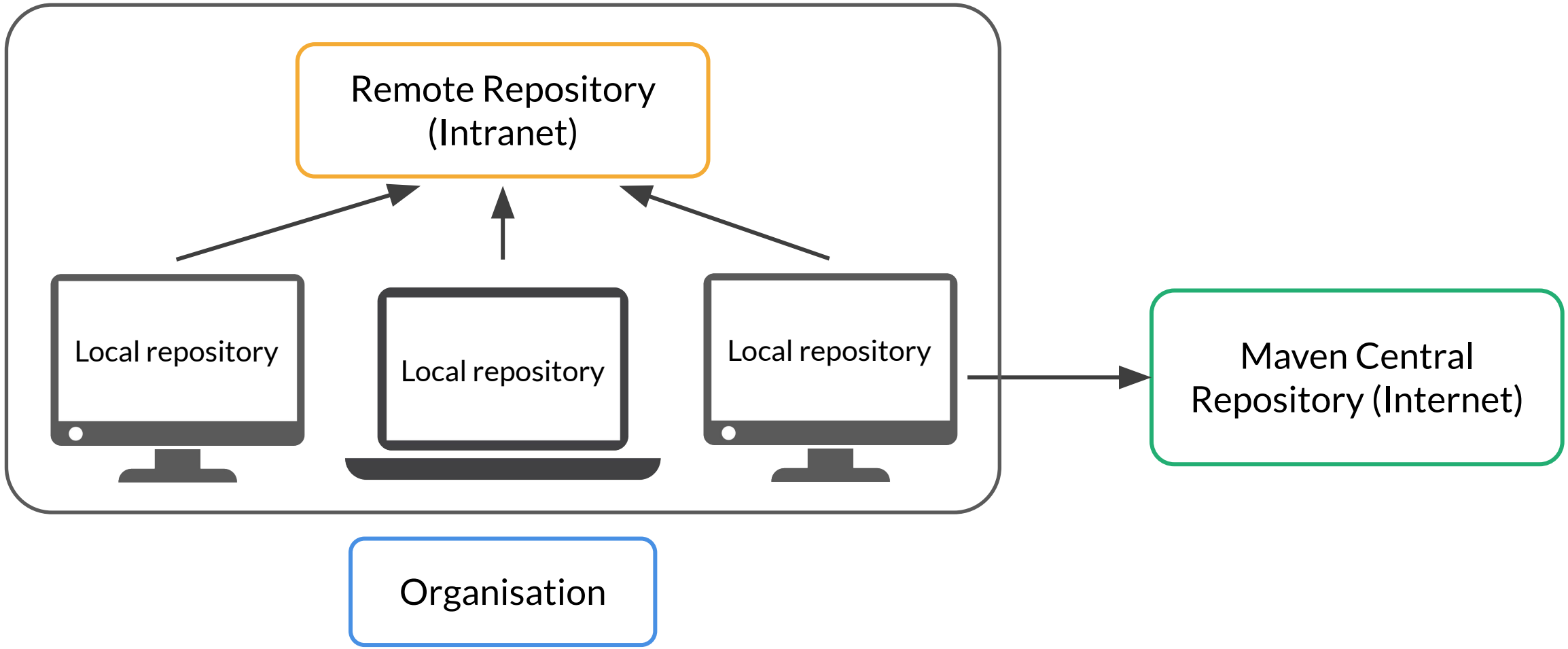
WHAT IS MAVEN?

- Maven is a software application build tool.
- It is primarily used to download .jar files.
- Other uses of Maven are as follows:
  - It provides a standard project structure.
  - It helps you in compiling, building and deploying a software.
  - It also helps you in generating the documentation for the source code.

API PLUGIN

- The external jars are also called **API plugins**, which is a software component that adds additional features to the existing applications.
- A JAR is a **set of Java classes, interface, enums and annotations** zipped into a .jar file.
- JAR files are mostly provided by **third-party vendor** companies.
- An API plugin needs to be **added separately** into the application classpath.
- Examples of API plugins are as follows:
  - The Spring Framework
  - JDBC Type 4 Driver

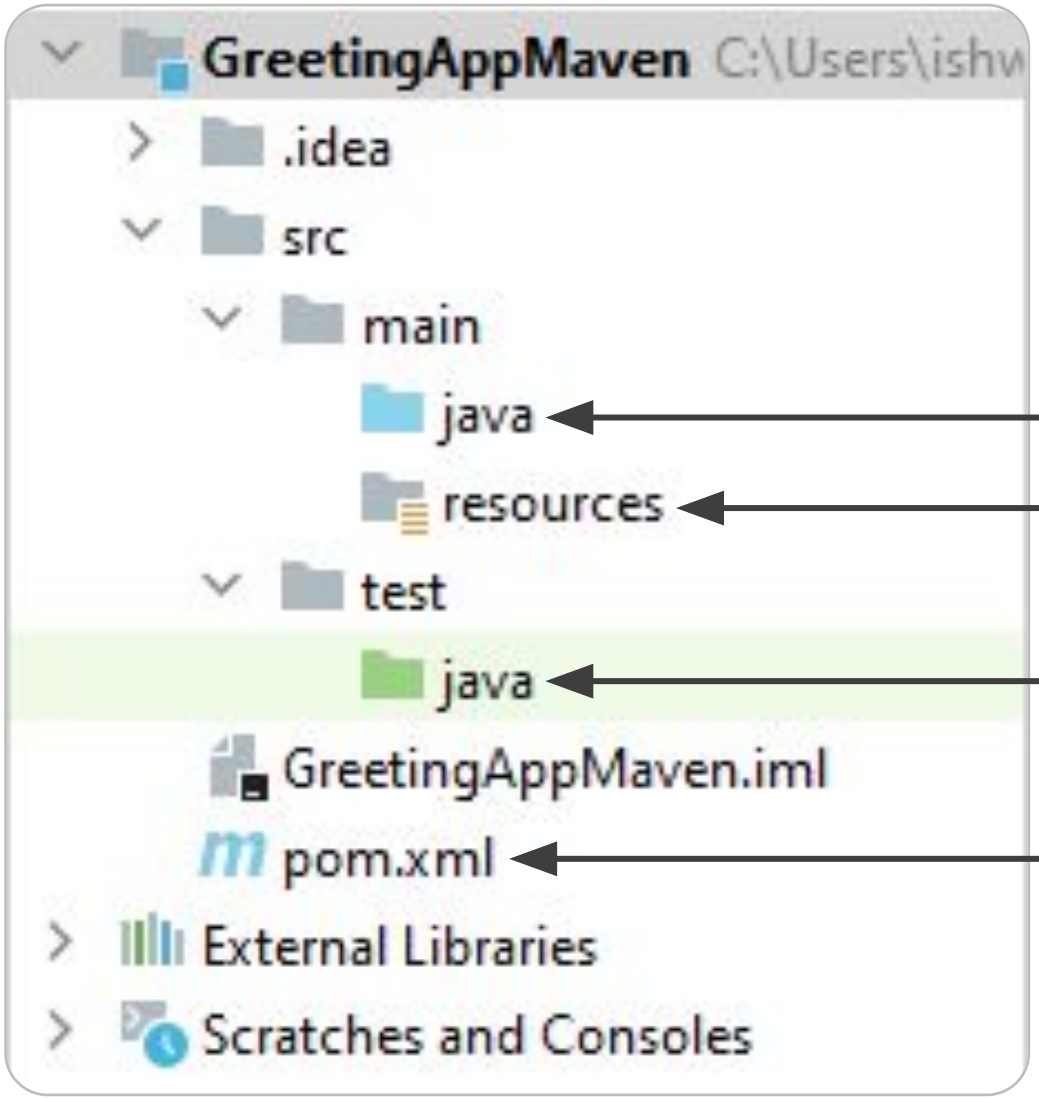
MAVEN REPOSITORIES



POM.XML

- It contains all the project-related configuration information that will be used by Maven to build an application.
- It also contains a list of all the API plugin jar dependencies that need to be downloaded.
- It resides in the base directory of a project.

PROJECT STRUCTURE



Contains the source code of your application

Contains the resources related to your source code, such as metadata, images and/or other files

Contains test cases for your application

Contains configuration information for your project

Bean Creation

BEAN CREATION

- Developers write the dependency classes. So, they need to tell the Spring about the classes that should be treated as dependency classes.
- They can provide this information to the Spring in the form of metadata (information about your code).
- There are three ways to provide metadata to the Spring, which are as shown below:
  - XML-based configuration
  - Java-based configuration
  - Annotations

BEAN CREATION

- **XML-based configuration:** You provide an XML file in which all the dependency classes are listed.
- **Java-based configuration:** You provide a type of factory class that contains the methods that return the objects of the dependency classes. The Spring Container executes these methods to instantiate the dependency classes. This factory class needs to be marked with the @Configuration annotation.
- **Annotation-based configuration:** Annotations are a way to provide metadata about your code. Annotations are similar to comments, which are not for other developers but for the tools that process your source code. Here, that tool is the Spring Framework.

Dependency Injection - Field Injection

DEPENDENCY INJECTION

- In Spring, you can perform DI in three ways:
  - Field or property injection
  - Setter injection
  - Constructor injection

DEPENDENCY INJECTION

- In Spring, DI is performed using the @Autowired annotation.
- Based on the way DI is performed in Spring, the @Autowired annotation can be applied over the following:
  - Fields or properties
  - Setter methods
  - Constructors

FIELD/PROPERTY INJECTION

```
@Component
public class EnglishGreetingService implements GreetingService {

  @Autowired
  private TimeService timeService;

  @Override
  public void greet(String name) {
     //Implementation goes here
  }
}
```

Dependency Injection - Setter Injection

SETTER INJECTION

```
@Component
public class EnglishGreetingService implements GreetingService {

  private TimeService timeService;

  @Autowired
  public void setTimeService(TimeService timeService) {
     this.timeService = timeService;
  }

  @Override
  public void greet(String name) {
     //Implementation goes here
  }
}
```

Dependency Injection - Constructor Injection

CONSTRUCTOR INJECTION

```
@Component
public class EnglishGreetingService implements GreetingService {

  private TimeService timeService;

  @Autowired
  public EnglishGreetingService(TimeService timeService) {
    this.timeService = timeService;
  }

  @Override
  public void greet(String name) {
    //Implementation goes here
  }
}
```

DEPENDENCY INJECTION

- Property and setter injection
  - To inject dependencies that are not critical for the dependent object to perform its functionalities
  - To inject dependencies that may change over time
- Constructor injection
  - As you cannot instantiate a class without providing all the arguments to the constructor, the constructor injection is used to inject those dependencies that are critical for the dependent object to perform its functionalities.

Bean Scope

BEAN SCOPE

- When you mark a class with the **@Component** annotation, the Spring container will treat that class as a dependency and instantiate that class to create a bean.
- By default, Spring creates one bean per dependency class, but you can control this feature using the **@Scope** annotation.

BEAN SCOPE

- The Spring framework supports the following five types of bean scopes:
  - Singleton
  - Prototype
  - Request
  - Session
  - Global-session
- Of these, three are applicable only in the case of a web-aware ApplicationContext.
- Note: In most cases, you will only deal with a **singleton** or a **prototype** bean scope.

BEAN SCOPE - SINGLETON

- It is the **default scope** of Spring bean inside the Spring IoC container.
- It is best described as **One Bean Per @Component Class Per IoC container**.
- The Spring IoC container gives the **same bean reference** every time when a request is made for a bean using the getBean() method or when a bean needs to be injected using the @Autowired annotation.
- This scope is used for **stateless beans** (which do not maintain any state).

BEAN SCOPE - SINGLETON

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
public class EnglishGreetingService implements GreetingService {
}
ApplicationContext context = new
AnnotationConfigApplicationContext("com.upgrad.greeting");
GreetingService greetingService1 =
(GreetingService)context.getBean("englishGreetingService");
GreetingService greetingService2 =
(GreetingService)context.getBean("englishGreetingService");


System.out.println(greetingService1 == greetingService2);
```

The output of the program would be “**true**”.

BEAN SCOPE - PROTOTYPE

- The Spring IoC container gives a **new bean instance reference** every time a request is made for a bean using the getBean() method or when a bean needs to be injected using the @Autowired annotation.
- Thus, a single bean with the Prototype scope can have any number of object instances.
- This scope is used for stateful beans (that maintain their state).

BEAN SCOPE - PROTOTYPE

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class EnglishGreetingService implements GreetingService {
}
ApplicationContext context = new
AnnotationConfigApplicationContext("com.upgrad.greeting");
GreetingService greetingService1 =
(GreetingService)context.getBean("englishGreetingService");
GreetingService greetingService2 =
(GreetingService)context.getBean("englishGreetingService");


System.out.println(greetingService1 == greetingService2);
```

The output of the program would be “**false**”.

BEAN SCOPE

- The other three scopes are available only for web-based applications, as shown here:
  - Request: The Spring container gives a single bean instance per HTTP request.
  - Session: The Spring container gives a single bean instance per HTTP session (user-level session).
  - Global-session: The Spring container gives a single bean instance per global HTTP session (application-level session).