Spring framework overview

Agenda

- Dependency injection: recap
- Spring framework basics
- Spring boot
- Spring framework stack

Dependency injection: recap

Dependency injection

- Dependency injection is a design pattern whereby dependencies of an object are provided (“injected”) externally
- Dependency injection is a form of IoC (Inversion of Control) as defined by the SOLID principles
- Dependency Injection (DI) is not the only form of IoC, another example is the template method pattern

Dependency injection

- Dependencies can be injected using different strategies:
  - setting a value on the field of a class directly (i.e. using reflection), also called **field injection**
  - using a setter method in the class, also called **setter injection**
  - by passing the dependency to the constructor of the class, also called **constructor injection**
  - by having the dependency inject itself to the object by having the object implement an interface with a setter method, also called **interface injection**

DI frameworks

- Dependency injection can be provided by a application utilities that are used to provide a mechanism for providing dependencies to object

- However since this is a common activity for many application a number of dependency injection frameworks provide this capability for applications

Dependency injection provides a fundamental mechanism for providing loose coupling between objects.

DI frameworks

- Widely-used dependency injection frameworks for Java applications are:
  - Spring framework
  - Google Guice
  - CDI/EJB (specifically in the context of JavaEE)
  - Dagger (for Android applications)

Spring framework basics

Spring DI

- Spring Core framework is a DI (dependency injection) container
- It is responsible to create and maintain objects and their dependencies
- All technologies from the Spring stack are built and make use of Spring Core framework

Spring context

- The entry point to the Spring framework is the Spring application context

- The application context manages the various objects used by the application through Spring (also called **beans**)

- The Spring context is bound to a particular mechanism for management of dependencies

Spring context

- Built-in mechanism for management of dependencies includes:
  - XML files (using ClassPathXmlApplicationContext instance)
  - annotations (using AnnotationConfigApplicationContext instance)

Spring context

- Spring DI framework can be supplied by the following dependencies:

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-core</artifactId>
	<version>${spring.version}</version>
</dependency>
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-context</artifactId>
	<version>${spring.version}</version>
</dependency>
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-beans</artifactId>
	<version>${spring.version}</version>
</dependency>
```

Spring XML context

```
ClassPathXmlApplicationContext context =
      new ClassPathXmlApplicationContext("context.xml");
ExampleService service =
      context.getBean(ExampleService.class);
service.someMethod();
context.close();
```

Spring XML context

context.xml

```
<?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/schem
a/context"
xsi:schemaLocation="http://www.springframework.org/schema/
beans
    http://www.springframework.org/schema/beans/spring-
beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-
context.xsd">
    <bean id="exampleservice“
       class="spring.examples.ExampleService">
    </bean>
</beans>
```

Spring annotation context

```
AnnotationConfigApplicationContext context = new
      AnnotationConfigApplicationContext("root.package");
ExampleService service =
      context.getBean(ExampleService.class);
service.someMethod();
context.close();
```

- To designate a bean class the @Component annotations can be used

```
@Component
public class ExampleService {

      public void someMethod() {
          …
      }
}
```

Spring annotation context

- Another way to retrieve a bean instance is to annotate a method in a @Configuration class with the @Bean annotation:

```
@Configuration
public class ServiceConfiguration {

      @Bean
      public ExampleService createExampleService() {
            return new ExampleService();
      }

}
```

Spring beans

- As a summary we can create Spring beans in any of the following ways:
  - through bean elements in XML configuration
  - through @Component-annotated classes
  - through @Bean-annotated factory methods

Spring XML and annotation-based configuration can be mixed but is generally not a good practice !

Spring beans

- Each Spring bean is represented by a BeanDefinition instance that provides metadata about the bean such as:
  - bean configuration (such as scope)
  - bean class and creation mechanism
  - bean dependencies
  - bean lifecycle

Spring bean scopes

- A bean scope defines how long does a Spring bean exist in a particular context:
  - singleton (default)
  - prototype
  - request
  - session
  - application
  - websocket

- Using annotation-based configuration a scope is specified with the @Scope annotation

Bean lifecycle

- The bean lifecycle can be customized by:
  - Implementing an interface such as InitializingBean (afterPropertiesSet() method) or DisposableBean (destroy() method)
  - Annotating methods with @PostConstruct and @PreDestroy annotations (preferable as it avoids coupling with Spring interfaces)

Bean post processing

- One or more bean post processors (instances of the **BeanPostProcessor** interface) can be defined to provide additional pre and post initialization logic

- The methods that need to be implemented are:
  - postProcessBeforeInitialization
  - postProcessAfterInitialization

Dependency injection

- A bean can be injected as a dependency of another bean using any of the following strategies:
  - constructor injection: passing the bean as an argument
  - setter/field injection: using a proper setter or reference as the injection target
  - factory method: passing the bean as an argument to a factory method that creates the instance

Dependency injection

- Dependency injection can be performed by means of the following annotations:
  - @Autowired (provided by Spring framework)
  - @Inject (provided by CDI)
  - @Resource (provided by Java SE)
- The bean to inject might be determined by type, name or qualifier

Component scan

- The packages used to scan for Spring components are supplied when the Spring context is created

- In addition root packages to be scanned for Spring components can be provided by the @ComponentScan annotation on a @Configuration-annotated class that is already scanned during context intialization

```
@Configuration
@ComponentScan(basePackages = “com.example")
public class Configuration  {
    // ...
}
```

Event handling

- The application context also provides publish and subscribe mechanism through events

- Beans that implement the ApplicationListener interface may consume triggered events

- Events are either:

  - built-in (such as **ContextStartedEvent** or **ContextStoppedEvent**)
  - custom

Application properties

- Different sources of application properties can be included for use by the application

- They can be specified in XML-based configuration using a **<property-placeholder>** element

- In annotation-based configuration property sources are specified with the **@PropertySource** annotations

Application properties

- Example:

```
@Configuration
@PropertySource("classpath:application.properties")
public class PropertiesWithJavaConfig {
    //...
}
```

- To inject a value from a property source the **@Value** annotation can be used

```
@Value( "${jdbc.url}" )
private String jdbcUrl;
```

By convention application.properties file is the default properties file used by Spring framework and does not need to be registered as a property source.

Application properties

- Application properties from property sources can also be retrieved by injecting an **Environment** instance

```
@Autowired
private Environment environment;
```

```
@Autowired
environment.getProperty("jdbc.url");
```

Spring boot

Spring boot overview

- Spring boot takes a step further in simplifying deployment and configuration of applications that use Spring framework

- In particular it provides capabilities such as:

  – automatic discovery of application.properties

  – automatic configuration of the Spring application (such as running an embedded Tomcat for deployment of @RestController classes)

Spring boot dependencies

- Spring boot is provided by any of the starter dependencies:
  - spring-boot-starter-web
  - spring-boot-starter-test
  - spring-boot-starter-jdbc
  - spring-boot-starter-data-jpa
  - spring-boot-starter-mail
  - a number of others

Spring boot application

- A typical Sprint Boot application entrypoint looks like the following:

```
@SpringBootApplication
public class Application {

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

Spring framework stack

Spring framework stack

- In earlier days Spring framework was simply a dependency injection container
- Nowadays it is a framework for building entire enterprise applications
- This is possible due to the large number of features provided by the additional components built around the Spring Core dependency injection container

Spring framework stack

https://start.spring.io/

Questions ?