Spring

Jaroslav Dytrych, David Kozák

Faculty of Information Technology Brno University of Technology
Božetěchova 1/2. 612 66 Brno - Královo Pole

dytrych@fit.vutbr.cz

7 November 2023

Spring

Contents

IoC (Inversion of Control)

- Some parts of our code receives the flow of control from a generic framework.
- Principle “Don´t call us, we´ll call you.”
- Instances of classes are not created but they are provided externally. Class do not need to know about implementation of an interface.
- Mostly used techniques are:
  - Constructor Injection – constructor is able to receive needed objects,
  - Setter Injection – class have a setters for necessary objects,
  - Interface Injection – interface with setters for necessary objects is defined, class have to implement this interface.
- Advantages:
  - Less dependencies between particular classes.
  - Explicitly specified dependencies.
  - Less type casting.
  - Better reusability of components.
- Disadvantages:
  - It is more complicated to understand the code.

What is Spring

- One of the most popular application development frameworks for enterprise Java.
- Spring makes JavaEE development easier.
- Benefits
  - Working with POJOs (no enterprise containers)
  - Modular framework
  - Can be combined with many other technologies
  - Framework for web with MVC
  - API for JDBC (Java Database Connectivity), Hibernate
  - IoC containers, dependency injection
  - Consistent transaction management

Dependency injection

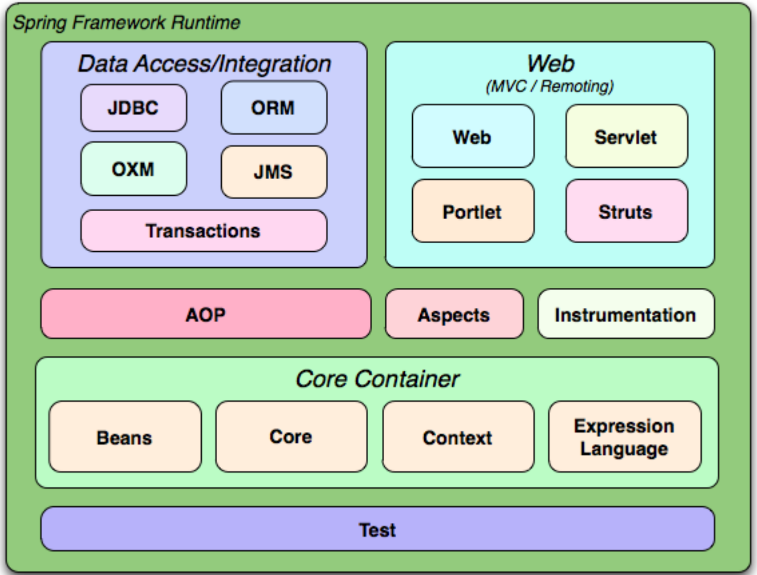
- Dependency injection is example of IoC.
- Makes writing of reusable and independent components possible.
  - Glues them together.
- Dependency injection in Spring can happen in the way of passing parameters to the constructor or by post-construction using setter methods.

Aspect oriented programming (AOP)

- Cross-cutting concerns are conceptually separate from the application´s business logic.
- Examples
  - Logging
  - Transaction management
  - Security
  - Caching
- Defining method interceptors and pointcuts
  - Pointcuts for events
  - Interceptors handles events

Spring architecture

Portlet – more servlets on the one page (generating fragments)



ORM – Object/Relational Mapping, OXM – Object/XML mapping

Core container

- The Core module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The Bean module provides `BeanFactory` which is a sophisticated implementation of the factory pattern.
- The Context module provides `ApplicationContext` interface, through which configured objects are accessible.
- The Expression Language module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data access/integration

- JDBC (Java Database Connectivity) abstraction layer.
- The ORM module provides integration layers for popular object-relational mapping APIs, including JPA (Java Persistence API), JDO (Java Data Objects), Hibernate, and iBatis.
- The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The JMS module contains features for producing and consuming messages.
- The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

Web

- Web
  - provides basic web-oriented integration features
  - Multipart fileupload
  - Initialization of IoC container
  - Servlet listeners, application context
- Web-Servlet module
  - Model-View-Controller (MVC)
- Web-Struts
  - Integration with Struts framework (MVC with strict rules)
  - deprecated as of Spring 3.0 – use Struts 2.0 and its Spring integration or Spring MVC solution instead
- Web-Portlet
  - MVC implementation to be used in portlet environment
  - Mirrors functionality of Servlet

Spring HelloWorld

- Create POJO object.
- Create Bean configuration file.
- Get application context.
    - `ClassPathXmlApplicationContext("Beans.xml");`
- Get POJO bean.
- Perform bean functionality.

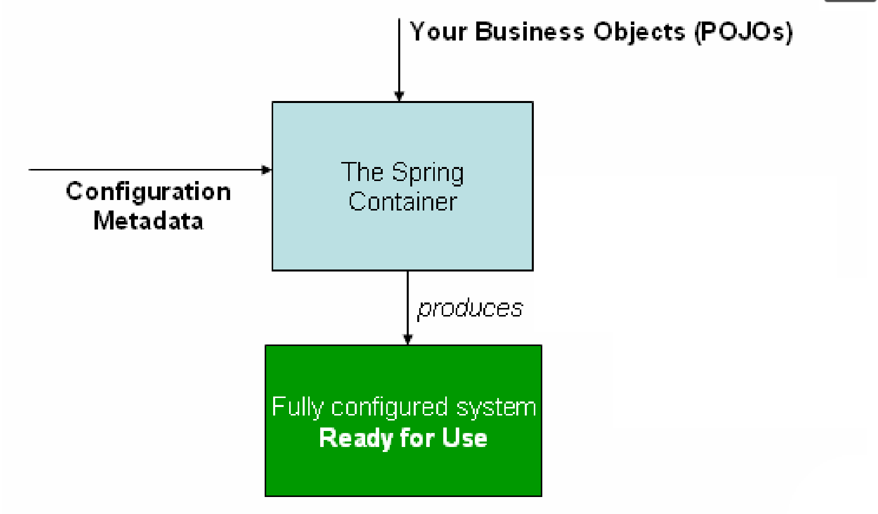
Spring HelloWorld

```java
public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");
        HelloWorld obj =
            (HelloWorld) context.getBean("helloWorld");
        obj.getMessage();
    }
}

public class HelloWorld {
    private String message;
    public void setMessage(String message){
        this.message = message;
    }
    public void getMessage(){
        System.out.println("Your Message : " + message);
    }
}
```

```xml
<bean id="helloWorld" class="cz.vutbr.fit.HelloWorld">
    <property name="message" value="Hello World!"/>
</bean>
```

Spring IoC containers



Spring IoC containers

- BeanFactory container
  - Basic support for dependency injection.
  - Class `XmlBeanFactory` (deprecated).
  - Various interfaces for backward compatibility.
- ApplicationContext container
  - More enterprise-specific functionality.
  - Wire beans together.
  - Recommended over BeanFactory.
  - `FileSystemXmlApplicationContext` – full path to XML
  - `ClassPathXmlApplicationContext` – XML on ClassPath
  - `WebApplicationContext` – for web applications (`WEB-INF`)

Spring bean definition

- Configuration metadata
  - How to create a bean,
  - Bean's lifecycle details,
  - Bean's dependencies.
- Properties
  - `class` – POJO class for the bean,
  - `name` – unique identifier,
  - `scope`,
  - `constructor-arg` – what should be passed into constructor,
  - `autowire` – Autowiring mode – injecting dependencies (nesting of dependencies),
  - `lazy-init` – Lazy-initialization mode – startup/request initialization,
  - `init-method` – initialization method,
  - `destroy-method` – destruction method.

Configuration metadata

Three types of configuration:

- XML based configuration file
  - `Beans.xml`

```
<!-- A bean definition with lazy init set on -->
<bean id="..." class="..." lazy-init="true">
</bean>
<!-- A bean definition with initialization method -->
<bean id="..." class="..." init-method="...">
</bean>
<!-- A bean definition with destruction method -->
<bean id="..." class="..." destroy-method="...">
</bean>
```

- Annotation based configuration
  - Used together with XML.
- Java based configuration
  - Avoid using XML (usage of Annotations, Java class replaces XML).

Example BeanInheritance

Spring bean scopes

- Singleton
  - one instance per Spring IoC (default),
  - we will get always same instance.
- Prototype
  - may have multiple instances,
  - always creates a new instance.
- Request
  - only valid in the context of web-aware Spring Aplication Context (HTTP request).
- Session
  - bean definition for a HTTP session.
- Global session
  - only in web applications, used with portlets.

Bean lifecycle

- Only two important callbacks
  - but other exists behind the scenes.
- Initialization callback
  - Default:
    ```
    void afterPropertiesSet() throws Exception;
    ```
  - Explicit setting:
    ```
    <bean id="exampleBean"
          class="examples.ExampleBean"
          init-method="init"/>
    ```
    - `public void init();`
- Destruction callback
  - `void destroy() throws Exception;`

  -
    ```
    <bean id="exampleBean"
          class="examples.ExampleBean"
          destroy-method="finish"/>
    ```
    - `public void finish();`

Bean Post-processors

- `BeanPostProcessor` interface defines callback methods
  - for own instantiation logic,
  - for dependency resolution logic.
- Methods:
  - `postProcessBeforeInitialization(Object bean, String beanName)`
  - `postProcessAfterInitialization(Object bean, String beanName)`
- Called after IoC instantiates a bean or an object.
- An `ApplicationContext` automatically detects any beans that are defined with implementation of the `BeanPostProcessor`.
- Order of interfaces execution can be specified.

Bean definition inheritance

- Bean definition contains a lot of configuration
    - constructor arguments,
    - property values,
    - container specific methods (`init`, `destroy`),
    - ...
- A child bean definition
    - inherits configuration data from a parent definition,
    - may override some methods and add others,
    - regular inheritance concept.
- XML configuration
    - attribute `parent`

Example BeanInheritance

Spring dependency injection

- Two types of dependency injection (DI)
  - Constructor based
  - Setter based
- Constructor based
  - Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.
- Setter based
  - Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate a bean.

Constructor based DI

- In the code

```
public TextEditor(SpellChecker spellChecker) {
    this.spellChecker = spellChecker;
}
```

- In the XML configuration file

```
<bean id="textEditor" class="com.tutorialspoint.TextEditor">
    <constructor-arg ref="spellChecker"/>
</bean>
<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
</bean>
```

- Constructor argument resolution
  - By name (of the parameter)

    ```
    <constructor-arg name="message" value="Test"/>
    ```

  - By type

    ```
    <constructor-arg type="java.lang.String" value="Test"/>
    ```

  - By index (in the parameters of the constructor)

    ```
    <constructor-arg index="0" value="Test"/>
    ```

Example ConstructorDependencyInjection

Setter based DI

- Setter based DI uses `property` tags.
- Attribude `ref` of property designates object to set.
- Regular form

```
<bean id="john-classic" class="com.example.Person">
    <property name="name" value="John Doe"/>
    <property name="spouse" ref="jane"/>
</bean>
<bean name="jane" class="com.example.Person">
    <property name="name" value="Jane Doe"/>
</bean>
```

- XML configuration using p-namespace

```
<bean id="john-classic" class="com.example.Person"
      p:name="John Doe"
      p:spouse-ref="jane"/>
</bean>
<bean name="jane" class="com.example.Person"
      p:name="Jane Doe"/>
</bean>
```

Example SetterDependencyInjection

Injecting Inner Beans

- Same model as inner classes in Java

```
<bean id="outerBean" class="...">
    <property name="target">
        <bean id="innerBean" class="..."/>
    </property>
</bean>
```

- Concretely

```
<bean id="textEditor" class="com.example.TextEditor">
    <property name="spellChecker">
       <bean id="spellChecker" class="com.example.SpellChecker"/>
    </property>
</bean>
```

Injecting collections

- Spring allows injecting following types of collections
  - List
    - list of values allowing duplicates
  - Set
    - set of values without duplicate
  - Map
    - injecting key-value pairs, where both can be of any type
  - Props
    - injecting key-value pairs, both of type `String`

Injecting collections examples

```
<property name="addressList">
<list>
  <value>Praha</value>
  <value>Ostrava</value>
  <value>Brno</value>
  <value>Brno</value>
</list>
</property>
```

```
<property name="addressSet">
<set>
  <value>Praha</value>
  <value>Ostrava</value>
  <value>Brno</value>
  <value>Brno</value>
</set>
</property>
```

```
<property name="addressMap">
<map>
  <entry key="1" value="Praha"/>
  <entry key="2" value="Ostrava"/>
  <entry key="3" value="Brno"/>
  <entry key="4" value="Brno"/>
</map>
</property>
```

```
<property name="addressProp">
<props>
  <prop key="one">Praha</prop>
  <prop key="two">Ostrava</prop>
  <prop key="three">Brno</prop>
  <prop key="four">Brno</prop>
</props>
</property>
```

- Results in calling `setAddressList`, `setAddressSet`, `setAddressMap` and `setAddressProp`

Example SpringCollections

Other injections

- Reference injections
  - Bean property value:
    ```
    <property name="myProperty">
        <ref bean="address1"/>
    </property>
    ```
  - Value of collection entry:
    ```
    <entry key="one" value-ref="address1"/>
    ```
- Injecting `null` value
  ```
  <bean id="..." class="exampleBean">
      <property name="email"><null/></property>
  </bean>
  ```
  - Resolved to `setEmail(null);`

Spring auto-wiring

- The Spring container can auto-wire relationships between collaborating beans.
- Auto-wiring modes
  - `no` – using explicit bean reference
  - `byName` – looks for property `name`
  - `byType` – only one bean of given type must exist
  - `constructor` – similar to `byType`, applies to constructor
  - `autodetect` – by default is tried by constructor, then by type

Examples SpringAutowireByName, SpringAutowireConstructor

Auto-wiring limitations

- Works best when used consistently across the project.

Limitations	**Description**
Overriding possibility	You can still specify dependencies using `<constructor-arg>` and `<property>` settings which will always override autowiring.
Primitive data types	You cannot autowire so-called simple properties such as primitives and Strings.
Confusing nature	Autowiring is less exact than explicit wiring, so if possible prefer using explicit wiring.

Annotation-based configuration

- Available since Spring 2.5.
- Performed before XML injection.
- Auto-wiring not turned on by default.
- Variants
  - `@Required` – on setter methods (deprecated)
  - `@Autowired` – on all methods
  - `@Qualifier` – together with `@Autowired` can specify bean to be injected
  - `@Resource`, `@PreDestroy`, `@PostConstruct`

Required is deprecated

- As of Spring 5.1, `@Required` is deprecated in favor of using constructor injection for required settings (or a custom `InitializingBean` implementation).
- You force clients to provide mandatory dependencies, making sure every object created is in a valid state after construction.
- You communicate mandatory dependencies publicly.
- Final fields also add to the immutable nature application components get. You can clearly distinguish between mandatory dependencies (final) and optional ones (non-final) usually injected through setter injection.

Annotation-based configuration

- `@Required` – affected bean property must be populated
  - otherwise `BeanInitializationException`
- `@Autowired` – autowiring `byType`
  - `@Autowired` with `required=false`
- `@Qualifier`
  - removes confusion (selection of particular bean)

    ```
    @Qualifier("student1")

    <bean id="student1" class="com.tutorialspoint.Student">
    ```
- `@Resource`
  - takes a name
  - provides `byName` autowiring
- `@PreDestroy` and `@PostConstruct`
  - callbacks

Examples springAnnotations, springAnnotationsAutowire, SpringAnnotationsAutowireConstructor, springQualifierAnnotation

Java based configuration

- Avoids using XML
  - `@Bean`
  - `@Configuration`

```
@Configuration
public class HelloWorldConfig {
    @Bean
    public HelloWorld helloWorld(){
        return new HelloWorld();
    }
}
```

- Is equivalent to XML annotation

```
<beans>
    <bean id="helloWorld" class="com.example.HelloWorld" />
</beans>
```

Example springConfiguration

Java based configuration

- Injecting bean dependencies

```
@Configuration
public class AppConfig {

    @Bean
    public Foo foo() {
         return new Foo(bar());
    }

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

Java based configuration

- `@Import` annotation
  - import from another configuration class

```
@Configuration
public class ConfigA {
    @Bean
    public A a() {
        return new A();
    }
}
```

```
@Configuration
@Import(ConfigA.class)
public class ConfigB {
    @Bean
    public B b() {
        return new B();
    }
}
```

- Life-cycle callbacks
  - `@Bean(initMethod = "init", destroyMethod = "cleanup" )`
- Specifying bean scope
  - `@Scope("prototype")`

Event handling in Spring

- Events are fired by `ApplicationContext`
  - `ContextStartedEvent` – when an `ApplicationContext` gets started.
  - `ContextRefreshedEvent` – when an `ApplicationContext` gets initialized or refreshed.
  - `ContextStoppedEvent`
  - `ContextClosedEvent`
  - `RequestHandledEvent`
    - Web-specific
- Implement appropriate interface
  - `ApplicationListener<...Event>`
- Override `onApplicationEvent(...Event)`

Example SpringEventHandling

Custom Events

- Event must inherit from `ApplicationEvent`.
- Class publishing a custom event must implement interface `ApplicationEventPublisherAware`
  - Contains `setApplicationEventPublisher()` for dependency on `ApplicationEventPublisher`.
- Class accepting custom event must implement interface `ApplicationListener<CustomEvent>`
  - method `onApplicationEvent(CustomEvent event)`
  - where `CustomEvent` is a name of custom event class.

Example SpringCustomEvent

Aspect oriented programming (AOP)

- AOP breaks program logic to separate "concerns".
- Spring provides a set of interceptors.
- Terminology
  - **Aspect** – module providing cross-cutting requirements (e.g. logging).
  - **Join point** – point, where AOP aspect can be "plugged in" (method execution, exception handling, changing object variable, etc.).
  - **Advice** – action (e.g. after method execution).
  - **Pointcut** – set of join points, where advice should be executed (a predicate that matches join points).
  - **Introduction** – allows adding new methods to existing classes.
  - Target object – object being adviced.
  - AOP proxy – an object created by the AOP framework in order to implement the aspect contracts.
  - Weaving – linking aspects with other application types or objects to create an advised object.

Advices

Advice	Description
before	Run advice before a method execution.
after	Run advice after a method execution regardless of its outcome.
after-returning	Run advice after a method execution only if method completes successfully.
after-throwing	Run advice after a method execution only if method exits by throwing an exception.
around	Run advice before and after the advised method is invoked.

XML Schema based aspects

- Declare an aspect
  - `<aop:aspect id="myAspect" ref="aBean">`
- Declare pointcut
  - `<aop:pointcut id="businessService" expression= "execution(* com.service.*.*(..))"/>`
- Declare advice
  - `<aop:before pointcut-ref="businessService" method="doRequiredTask"/>`

```
<aop:config>
  <aop:aspect id="log" ref="logging">
    <aop:pointcut id="selectAll"
                  expression="execution(* com.example.*.*(..))"/>
    <aop:before pointcut-ref="selectAll" method="beforeAdvice"/>
    <aop:after pointcut-ref="selectAll" method="afterAdvice"/>
    <aop:after-returning pointcut-ref="selectAll"
                         returning="retVal"
                         method="afterReturningAdvice"/>
    <aop:after-throwing pointcut-ref="selectAll" throwing="ex"
                        method="AfterThrowingAdvice"/>
  </aop:aspect>
</aop:config>
```

Example SpringAspect

AspectJ based aspects

- Enabled by
  - `<aop:aspectj-autoproxy/>`
- Declaring aspect

```java
@Aspect
public class AspectModule {
...
}
```

- Declare pointcut

```java
@Pointcut("execution(* com.example.Student.*(..))")
private void businessService() {}
```

- Declare advice

```java
@Around("businessService()")
public void doAroundTask(){
...
}
```

- It is possible to access pointcut information from advice

```java
@Before("businessService()")
public void beforeAdvice(JoinPoint jp){
  System.out.println("Going to " + jp.getSignature());
}
```

Example springJAspect

Spring JDBC framework

- Spring takes care of low-level things.
- The only necessary actions
  - define connection,
  - specify SQL statement.
- JDBC template class `JdbcTemplate`
  - executes SQL queries,
  - returns `ResultSet`,
  - threadsafe once configured,
  - can be injected to multiple DAOs (Data access objects).
- `RowMapper<T>`
  - mapping rows of a `ResultSet` on a per-row basis

```
public class StudentMapper implements RowMapper<Student> {
  @Override
  public Student mapRow(ResultSet rs, int rowNum)
      throws SQLException {
    Student student = new Student();
    student.setId(rs.getInt("id"));
    student.setName(rs.getString("name"));
    student.setAge(rs.getInt("age"));
    return student;
  }
}
```

Examples SpringJDBC, SpringJDBCProcedure

Configuring datasource

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
  <property name="url"
value="jdbc:mysql://localhost:3306/TEST?serverTimezone=Europe/Prague"
/>
  <property name="username" value="springJDBC"/>
  <property name="password" value="password"/>
</bean>
```

- Data object
  - read/write database
  - support for JDBC, Hibernate, JPA, JDO

Executing SQL queries

- Query for object (`Integer`)

```
String SQL = "select count(*) from Student";
int rowCount = jdbcTemplateObject.queryForObject(SQL,
               Integer.class);
```

- Insert

```
String SQL = "insert into Student (name, age) values (?, ?)";
jdbcTemplateObject.update(SQL, new Object[]{"Sue", 11});
```

- Update

```
String SQL = "update Student set name = ? where id = ?";
jdbcTemplateObject.update(SQL, new Object[]{"Ed", 10});
```

- Delete

```
String SQL = "delete Student where id = ?";
jdbcTemplateObject.update(SQL, new Object[]{20});
```

Spring Transaction Management

- Local transactions
  - specific to a single transactional resource like a JDBC connection.
- Global transactions
  - distributed computing environment.
- Transactions in Spring
  - programmatic,
  - declarative.

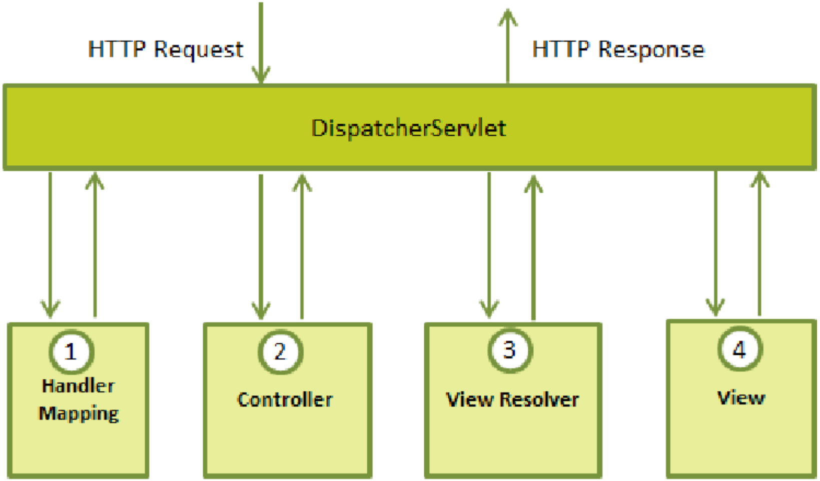
Examples SpringTransaction, SpringTransactionDeclarative

Spring Web MVC framework

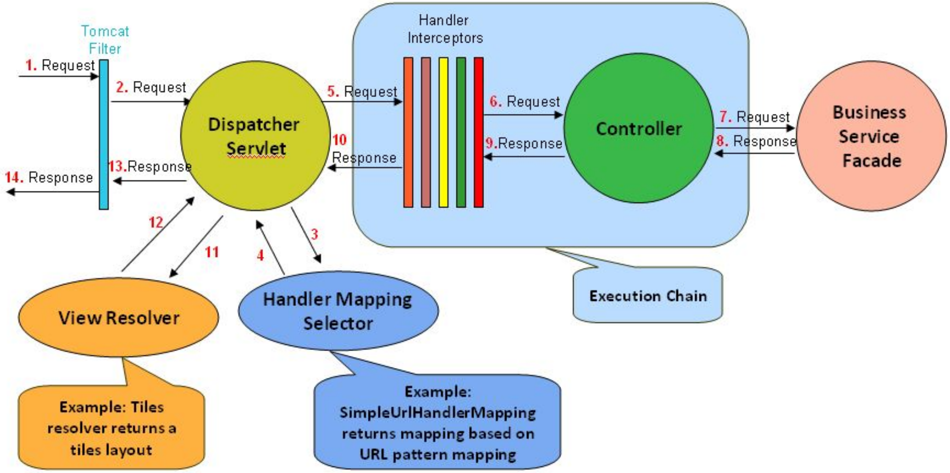
- The Model encapsulates the application data and in general they will consist of POJO.
- The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- The Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

Dispatcher servlet

- Handles HTTP requests and responses



Dispatcher servlet



Dispatcher servlet

- After receiving an HTTP request, `DispatcherServlet` consults the `HandlerMapping` to call the appropriate Controller.
- The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the `DispatcherServlet`.
- The `DispatcherServlet` will take help from `ViewResolver` to pick up the defined view for the request.
- Once view is finalized, the `DispatcherServlet` passes the model data to the view which is finally rendered on the browser.

Configuration

- Declare servlet in `web.xml`.
- Customize it in `<appName>-servlet.xml`.
- Specify URL pattern to be handled (`*.jsp`).
- Enable Spring MVC annotation scanning.
  - `<context:component-scan...>`
- Define controller and mapping.

Defining controller

- The `@Controller` annotation defines the class as a Spring MVC controller.
- `@RequestMapping` annotation is used to map a URL to either an entire class or a particular handler method.
- Due to return value, redirection to `hello.jsp` will happen.

```
@Controller
public class HelloController {
  @RequestMapping(value = "/hello", method = RequestMethod.GET)
  public String printHello(ModelMap model) {
    model.addAttribute("message",
      "Hello Spring MVC Framework!");
    return "hello";
  }
}
```

Creating JSP views

- Here `${message}` is the attribute which we have setup inside the Controller. You can have multiple attributes to be displayed inside your view.
- File path will be `/WEB-INF/hello/hello.jsp`

```
<html>
    <head>
        <title>Hello Spring MVC</title>
    </head>
    <body>
        <h2>${message}</h2>
    </body>
</html>
```

Examples HelloWeb, SpringMVC

Spring Security

- Some pages should not be publicly available.
- Managing user roles
  - Users can view pages according to their user role level (user, admin, super-admin).
- Login
  - Default
    - Custom appearance may be defined.
  - HTTP
    - HTTP Authentication (RFC 7235, 7615, 7616, 7617).
    - User logged in as long as browser runs.

Spring security configuration

- XML based
  - `web.xml`
  - `mvc-dispatcher-servlet.xml`
  - `spring-security.xml`
- Annotation based
  - `@Configuration`
  - `@EnableWebSecurity`
  - `@EnableWebMVC` – imports the Spring MVC configuration
  - `@ComponentScan`

XML configuration

- spring-security.xml

```
<http auto-config="true">
  <intercept-url pattern="/admin**" access="ROLE_USER" />

  <form-login
    login-page="/login"
    default-target-url="/welcome"
    authentication-failure-url="/login?error"
    username-parameter="username"
    password-parameter="password" />
  <logout logout-success-url="/login?logout"  />
  <!-- enable csrf protection -->
  <csrf/>
</http>

<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="user" password="123456"
            authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

Example SpringCustomSecurity

XML configuration from 5.0.0.RC1

- From version 5.0.0.RC1 `password-encoder` must be set.

```
<authentication-manager>
  <authentication-provider>
    <password-encoder ref="passwordEncoder" />
    <user-service>
      <user name="user" password="123456"
            authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>

<b:bean id="textEncryptor"
  class="org.springframework.security.crypto.encrypt.Encryptors"
  factory-method="noOpText" />

<b:bean id="passwordEncoder"
  class=
"org.springframework.security.crypto.password.NoOpPasswordEncoder"
  factory-method="getInstance" />
```

Example SpringCustomSecurity5

Annotation configuration

```
@Configuration
@EnableWebSecurity
public class AppSecurityConfig extends WebSecurityConfigurerAdapter {

  @Autowired
  public void configureGlobal(AuthenticationManagerBuilder auth)
      throws Exception {
    auth.inMemoryAuthentication().withUser("tom").
      password("123456").roles("USER");
    auth.inMemoryAuthentication().withUser("bill").
      password("123456").roles("ADMIN");
    auth.inMemoryAuthentication().withUser("james").
      password("123456").roles("SUPERADMIN");
  }

  @Override
  protected void configure(HttpSecurity http) throws Exception {

    http.authorizeRequests()
      .antMatchers("/protected/**").access("hasRole('ROLE_ADMIN')")
      .antMatchers("/confidential/**").access("hasRole('ROLE_SUPERADMIN')")
      .and().formLogin();

  }
}
```

Example SpringSecurityAnnotations

Annotation configuration from 5.0.0.RC1

```
auth.inMemoryAuthentication().withUser("tom").
  password("{noop}123456").roles("USER");
```

Example SpringSecurityAnnotations5

Password hashing

- Supported password formats
  - `plaintext`
  - `sha,sha256`
  - `md4,md5`
  - Integration with LDAP is possible.
  - Newer versions supports also PBKDF2 (Password-Based Key Derivation Function 2), BCrypt, sCrypt, Argon2, . . .

```
<authentication-manager>
  <authentication-provider>
    <password-encoder hash="sha" />
    <user-service>
      <user name="user"
            password="7c4a8d09ca3762af61e59520943dc26494f8941b"
            authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

CSRF exploit

- Cross-site request forgery
  - Spring has to know, that the origin of the request to the admin section is from the trusted source (the form comes from this application and not from an attacker who sends a request for example from an advertisement).
  - Otherwise hackers would be able to do exploit when the admin is logged in.

- Post CSRF token in request

```
<form action="$logoutUrl" method="post" id="logoutForm">
  <input type="hidden" name="$_csrf.parameterName"
         value="$_csrf.token" />
</form>
```

- Enable CSRF protection in configuration
  - `<csrf/>`

Database login

Xml

```
<authentication-manager>
  <authentication-provider>
    <jdbc-user-service data-source-ref="dataSource"
      users-by-username-query=
        "select username,password,enabled from users where username=?"
      authorities-by-username-query=
        "select username,role from user_roles where username =?"/>
 </authentication-provider>
</authentication-manager>

<bean id="dataSource" class=
    "org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
  <property name="url"
value="jdbc:mysql://localhost:3306/test?serverTimezone=Europe/Prague"
/>
  <property name="username" value="springJDBC" />
  <property name="password" value="password" />
</bean>
```

Database login

Annotation

```
@Autowired
DataSource dataSource;
@Autowired
public void configAuthentication(AuthenticationManagerBuilder auth)
    throws Exception {
  auth.jdbcAuthentication().passwordEncoder(passwordEncoder())
    .dataSource(dataSource)
    .usersByUsernameQuery(
      "select username,password,enabled from users where username=?")
    .authoritiesByUsernameQuery(
      "select username,role from user_roles where username=?");
}

@Bean
public PasswordEncoder passwordEncoder() {
  return NoOpPasswordEncoder.getInstance();
}
```

Example SpringDatabaseLogin

Spring Boot

- A preferred way how to create Spring applications
- Create stand-alone Spring apps
- Application server is embedded
- Opinionated starter dependencies
  - *"Convention over configurations"*
- No XML configuration
- Provides a starter page for convenience
  - `https://start.spring.io`

Spring Boot Apps - Typical Archictecture

- Controllers
  - Handle all HTTP related tasks
  - Should contain no business logic
- Services
  - The business logic should be here
  - Can contact other services or update the database via repositories
- Repositories
  - Essentially DAOs
  - The code is mostly created automatically for us by Spring Data

References

- Spring tutorial
  - http://www.tutorialspoint.com/spring/index.htm
- Spring documentation
  - https://docs.spring.io/spring-framework/docs/
  - https://docs.spring.io/spring-framework/docs/5.3.x/
  - https://docs.spring.io/spring-framework/docs/5.2.9.RELEASE_to_5.2.10.RELEASE/
- Using Spring Handler Interceptors to Decouple Business Logic
  - https://sdevarapalli.wordpress.com/2011/02/16/using-spring-handler-interceptors-to-decouple-business-logic/
- Overview of Spring MVC Architecture
  - http://terasolunaorg.github.io/guideline/1.0.1.RELEASE/en/Overview/SpringMVCOverview.html
- Spring MVC Framework
  - http://www.wideskills.com/spring/spring-mvc-framework

References

- AOP
  - http://www.mkyong.com/spring/spring-aop-example-pointcut-advisor/
  - http://www.byteslounge.com/tutorials/spring-aop-pointcut-advice-example
- Logout in Spring Security 4
  - http://websystique.com/spring-security/spring-security-4-logout-example/
- Others
  - https://stackoverflow.com/questions/26515700/mysql-jdbc-driver-5-1-33-time-zone-issue
  - https://github.com/spring-projects/spring-framework/wiki/Upgrading-to-Spring-Framework-5.x#data-access-and-transactions
  - https://www.baeldung.com/java-varargs
  - https://stackoverflow.com/questions/53358568/springs-annotation-type-required-deprecation
  - https://odrotbohm.de/2013/11/why-field-injection-is-evil/

Thank you for your attention!