Spring framework notes

Spring core

Why spring?

- Very popular framework for building Java applications
- Initially, a simpler and lightweight alternative to J2EE
- Provides large number of helper classes … makes things easier

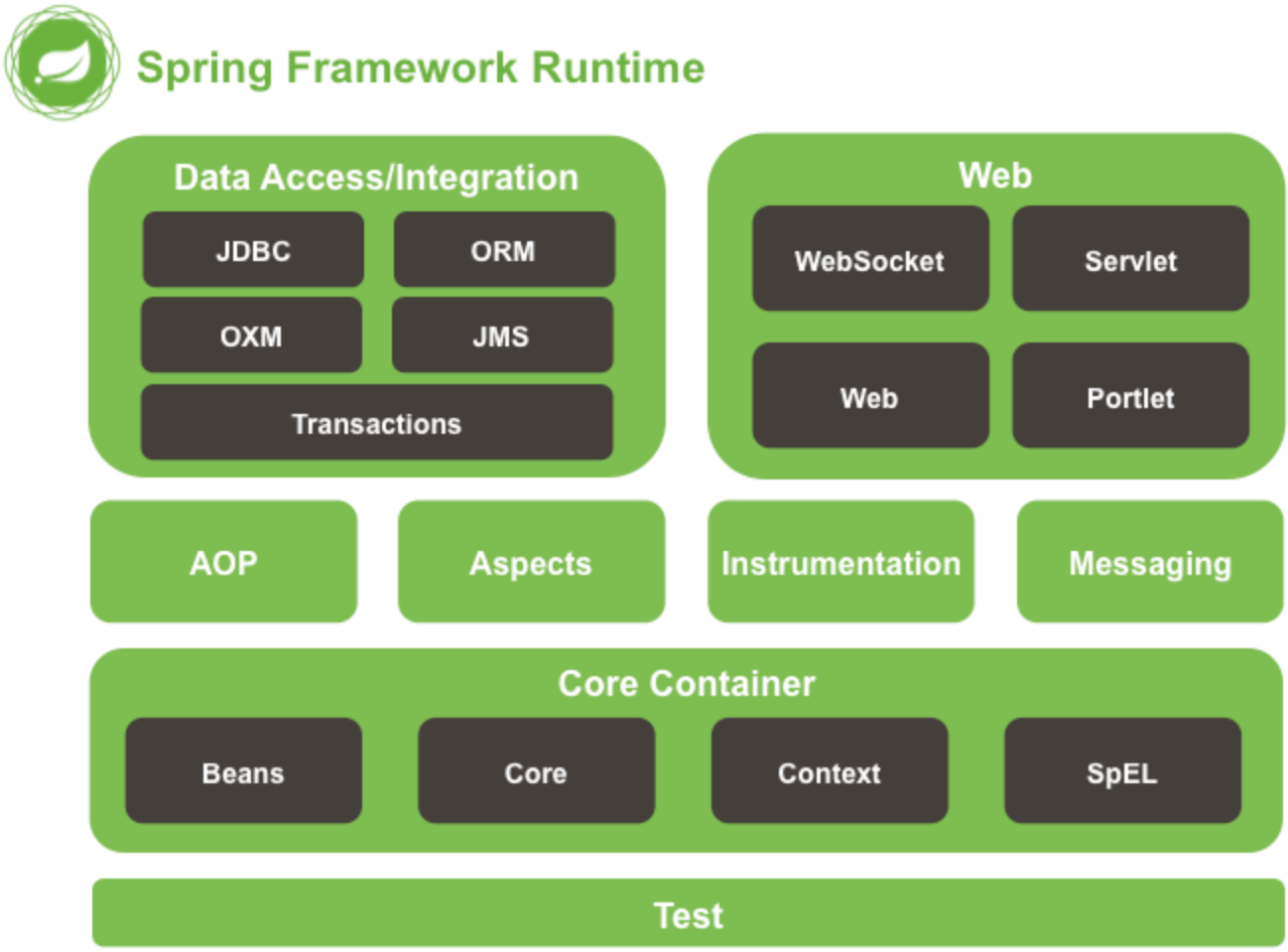
Goals of spring

- Lightweight development with Java POJOs (Plain-Old-Java-Objects)
- Dependency injection to promote loose coupling
- Declarative programming with Aspect-Oriented-Programming (AOP)
- Minimize boilerplate Java code

Spring core framework modules

- **Spring core container**
  - The *Core Container* consists of the spring-core, spring-beans, spring-context, spring-context-support, and spring-expression (Spring Expression Language) modules.
  - The spring-core and spring-beans modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.
  - The *Context* (spring-context) module builds on the solid base provided by the *Core and Beans* modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The ApplicationContext interface is the focal point of the Context module. spring-context-support provides support for integrating common third-party libraries into a Spring application context for caching (EhCache, Guava, JCache), mailing (JavaMail), scheduling (CommonJ, Quartz) and template engines (FreeMarker, JasperReports, Velocity).
  - The **spring-expression** module provides a powerful *Expression Language* for querying and manipulating an object graph at runtime. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the content of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container.

- **AOP and Instrumentation**
  - The **spring-aop** module provides an *AOP* Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
  - The separate **spring-aspects** module provides integration with AspectJ.
  - The **spring-instrument** module provides class instrumentation support and classloader implementations to be used in certain application servers. The spring-instrument-tomcat module contains Spring's instrumentation agent for Tomcat.
- **Messaging**
  - **spring-messaging** module provides key abstractions from the *Spring Integration* project such as Message, MessageChannel, MessageHandler, and others to serve as a foundation for messaging-based applications
- **Data Access/Integration**
  - The **spring-jdbc** module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.
  - The **spring-tx** module supports programmatic and declarative transaction management for classes that implement special interfaces and for *all your POJOs (Plain Old Java Objects)*.
  - The **spring-orm** module provides integration layers for popular object-relational mapping APIs, including JPA and Hibernate. Using the spring-orm module you can use these O/R-mapping frameworks in combination with all the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.
  - The **spring-oxm** module provides an abstraction layer that supports Object/XML mapping implementations such as JAXB, Castor, JiBX and XStream.
  - The **spring-jms** module (Java Messaging Service) contains features for producing and consuming messages. Since Spring Framework 4.1, it provides integration with the spring-messaging module.
- **Web layer**
  - The **spring-web** module provides basic web-oriented integration features such as multipart file upload functionality and the initialization of the IoC container using Servlet listeners and a web-oriented application context. It also contains an HTTP client and the web-related parts of Spring's remoting support.
  - The **spring-webmvc** module (also known as the *Web-Servlet* module) contains Spring's model-view-controller (*MVC*) and REST Web Services implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms and integrates with all the other features of the Spring Framework.
  - The **spring-webmvc-portlet** module (also known as the *Web-Portlet* module) provides the MVC implementation to be used in a Portlet environment and mirrors the functionality of the Servlet-based spring-webmvc module.
- **Test**
  - The **spring-test** module supports the unit testing and integration testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
WebSocket
Servlet
Web
Portlet
AOP
Aspects
Instrumentation
Messaging
Core Container
Beans
Core
Context
SpEL
Test

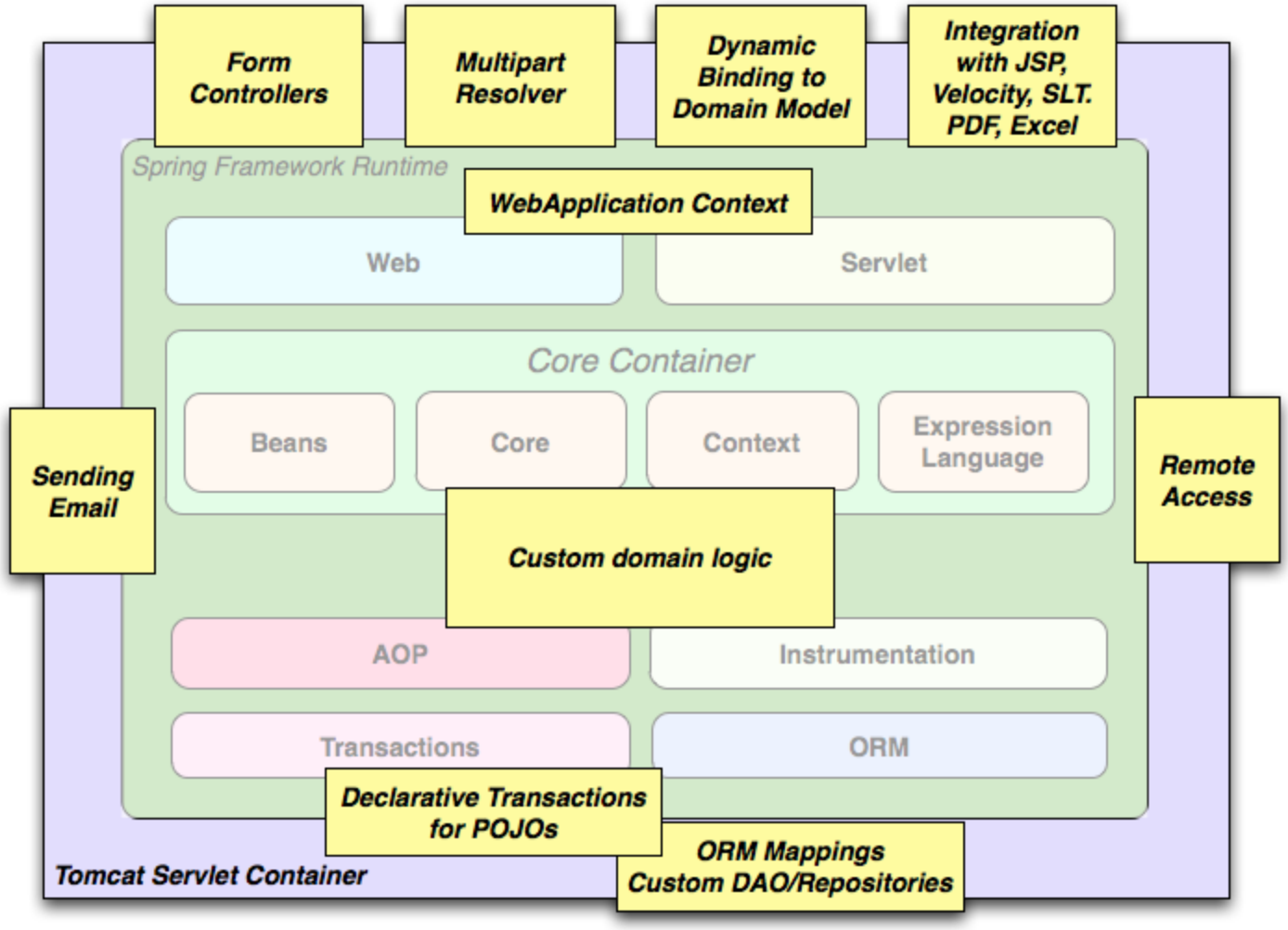


Figure 2. Typical full-fledged Spring web application

Spring framework artefacts

Group Id is org.springframework for all of the below artefacts

ArtifactId	Description
ArtifactId	Description
spring-aop	Proxy-based AOP support
spring-aspects	AspectJ based aspects
spring-beans	Beans support, including Groovy
spring-context	Application context runtime, including scheduling and remoting abstractions
spring-context-support	Support classes for integrating common third-party libraries into a Spring application context
spring-core	Core utilities, used by many other Spring modules
spring-expression	Spring Expression Language (SpEL)

spring-instrument	Instrumentation agent for JVM bootstrapping
spring-instrument-tomcat	Instrumentation agent for Tomcat
spring-jdbc	JDBC support package, including DataSource setup and JDBC access support
spring-jms	JMS support package, including helper classes to send/receive JMS messages
spring-messaging	Support for messaging architectures and protocols
spring-orm	Object/Relational Mapping, including JPA and Hibernate support
spring-oxm	Object/XML Mapping
spring-test	Support for unit testing and integration testing Spring components
spring-tx	Transaction infrastructure, including DAO support and JCA integration
spring-web	Foundational web support, including web client and web-based remoting
spring-webmvc	HTTP-based Model-View-Controller and REST endpoints for Servlet stacks
spring-websocket	WebSocket and SockJS infrastructure, including STOMP messaging support

Spring container

Primary functions
• Create and manage objects (Inversion of Control)
• Inject object's dependencies (Dependency Injection)

Inversion of control

The approach of outsourcing the construction and management of objects / beans

What is a bean?

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

Different ways to configure spring container

• XML configuration file (legacy, but most legacy apps still use this) (ClassPathXmlApplicationContext)
• Java Annotations (modern)
• Java Source Code (modern)

Spring Development Process

1. Configure your Spring Beans
2. Create a Spring Container
3. Retrieve Beans from Spring Container

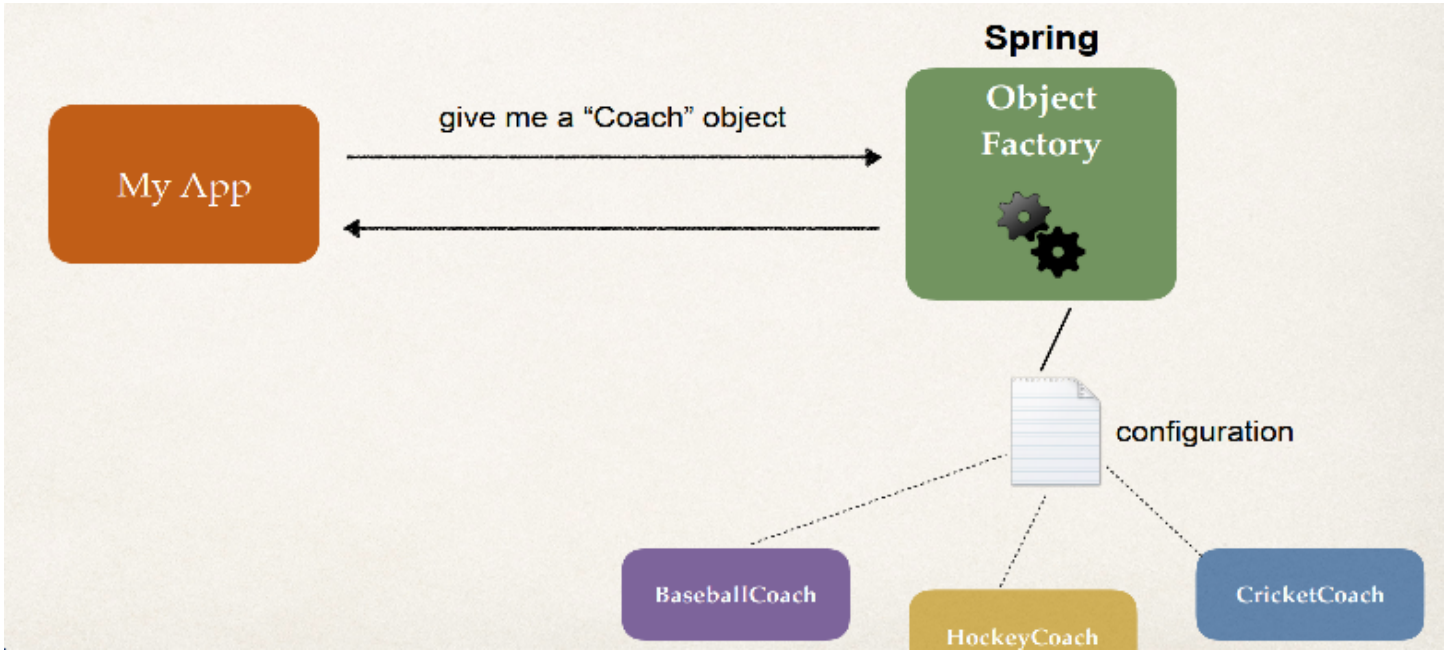
File: applicationContext.xml

```
<beans ... >

    <bean  id="myCoach"
        class="com.luv2code.springdemo.BaseballCoach">
    </bean>

</beans>
```

```
// create a spring container
ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext("applicationContext.xml");

// retrieve bean from spring container
Coach theCoach = context.getBean("myCoach", Coach.class);
```

File: applicationContext.xml

```
<bean  id="myCoach"
    class="com.luv2code.springdemo.BaseballCoach">
</bean>
```



Dependency injection

The dependency inversion principle: The client delegates to calls to another object the responsibility of providing its dependencies.

Dependency injection (DI) is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes, or the *Service Locator* pattern.

Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies, and does not know the location or class of the dependencies. As such, your classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

DI exists in two major variants, Constructor-based dependency injection and Setter-based dependency injection.

Constructor and setter injection using xml configuration. Note that there are different ways to inject parameters by name, by index etc and also to inject literal values, arrays, maps, values from property files etc. Refer Spring Framework for more details

```
<?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"
   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">
   <!-- Define your beans here -->
   <!-- define the dependency -->
   <bean id="myFortuneService" class="com.luv2code.springdemo.HappyFortuneService">
   </bean>

   <bean id="myCoach"
           class="com.luv2code.springdemo.TrackCoach">
           <!-- set up constructor injection -->
           <constructor-arg ref="myFortuneService" />
   </bean>

   <bean id="myCricketCoach"
       class="com.luv2code.springdemo.CricketCoach">
       <!-- set up setter injection -->
       <property name="fortuneService" ref="myFortuneService" />
```

```
    </bean>
</beans>
```

Constructor-based or setter-based DI?

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for *mandatory dependencies* and setter methods or configuration methods for *optional dependencies*. Note that use of the @Required annotation on a setter method can be used to make the property a required dependency.

The Spring team generally advocates constructor injection as it enables one to implement application components as *immutable objects* and to ensure that required dependencies are not null. Furthermore, constructor-injected components are always returned to client (calling) code in a fully initialized state.

Bean scope

Scope refers to the lifecycle of a bean. It determines below things
• How long does the bean live?
• How many instances are created?
• How is the bean shared?

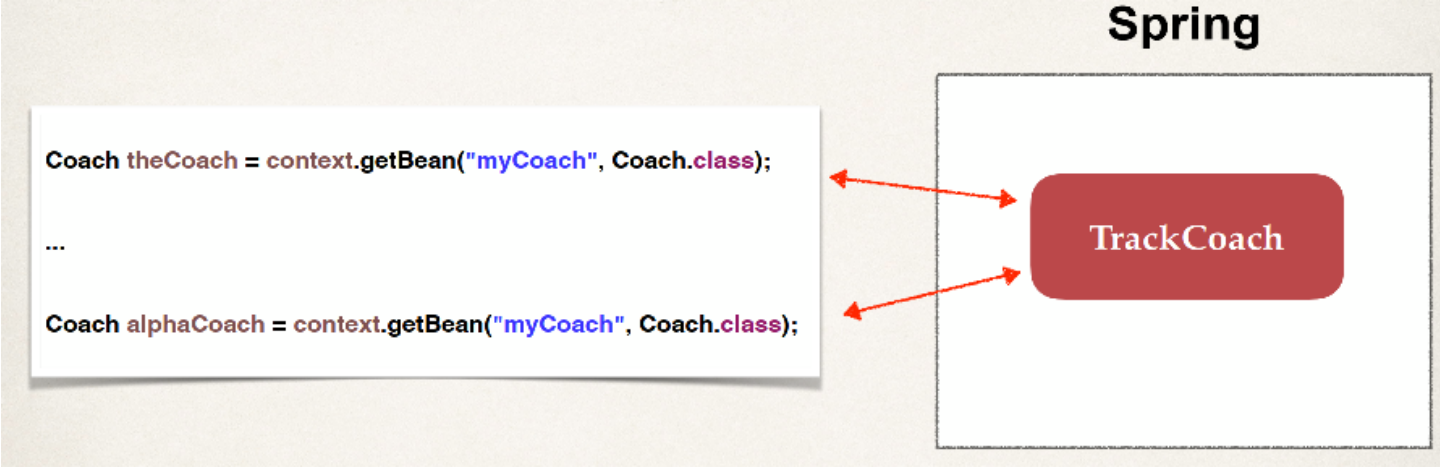
List of scopes a bean can have

singleton	Create a single shared instance of the bean. **Default scope**. Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Creates a new bean instance for each container request. Scopes a single bean definition to any number of object instances. Separate instances are received on each bean retrieval from container. For "prototype" scoped beans, **Spring does not call the destroy method**.
request	Scoped to an HTTP web request. Only used for web apps. Scopes a single bean definition to the lifecycle of a single HTTP request, that is, each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scoped to an HTTP web session. Only used for web apps. Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext (spring-mvc).
websocket	Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.
global-session	Scoped to a global HTTP web session. Only used for webapps. Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

What is singleton?

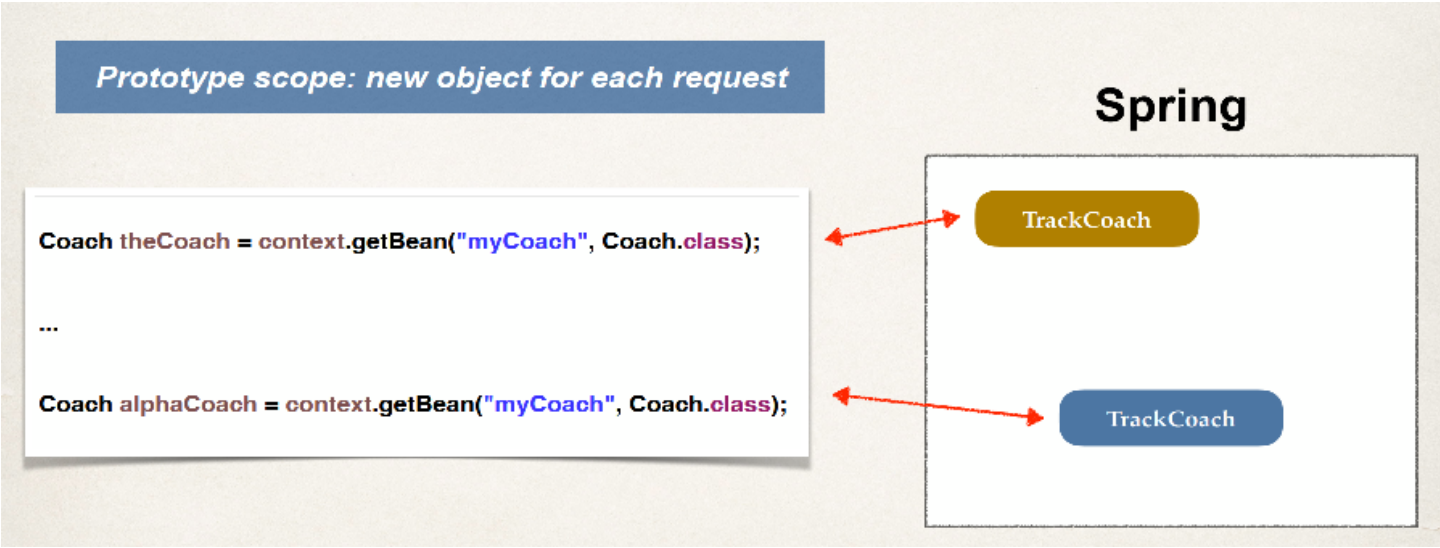
Spring Container creates only one instance of the bean, by default
• It is cached in memory
• All requests for the bean
• will return a SHARED reference to the SAME bean

In the below picture, the same Java object will be injected in both places. So “theCoach” and “alphaCoach” refer to the same Java object

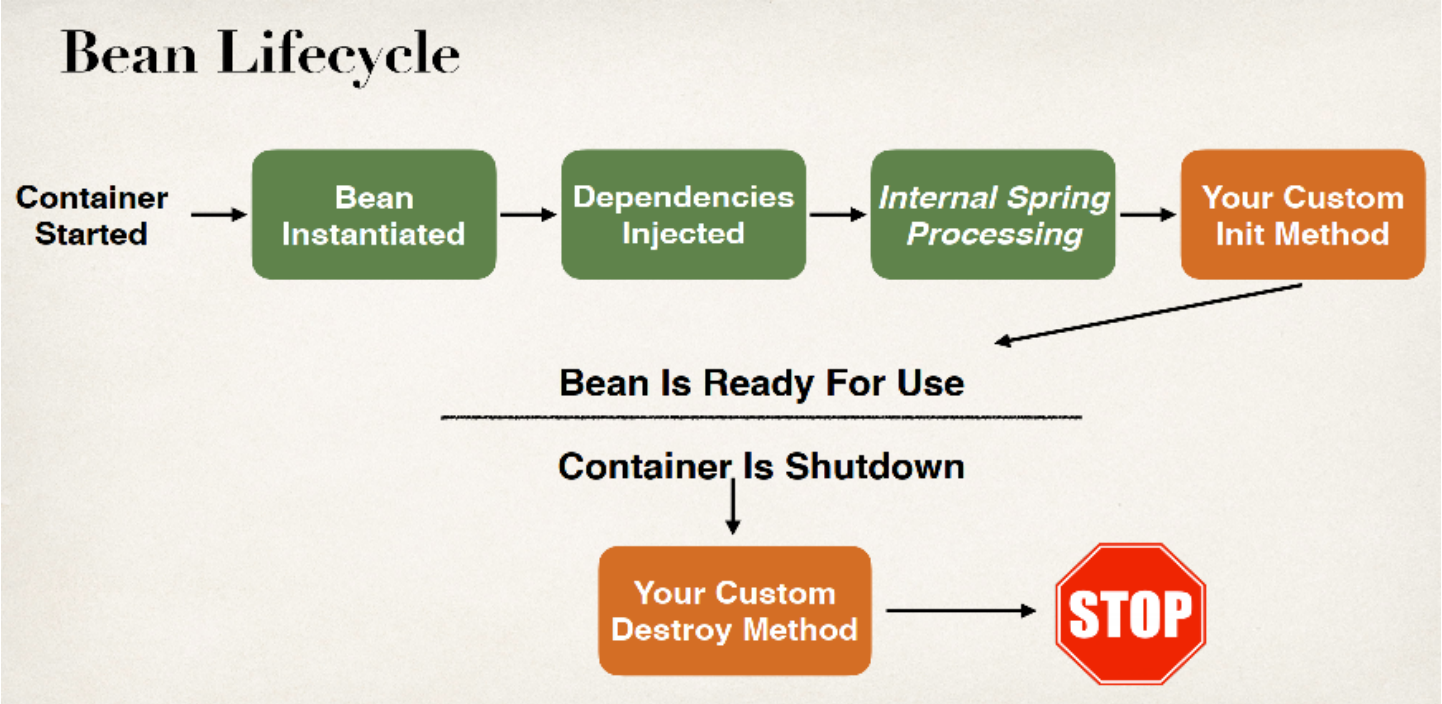


Prototype scope

```
<beans >
	<bean id="myCoach"
		class="com.luv2code.springdemo.TrackCoach"
		scope="prototype">
	</bean>
</beans>
```



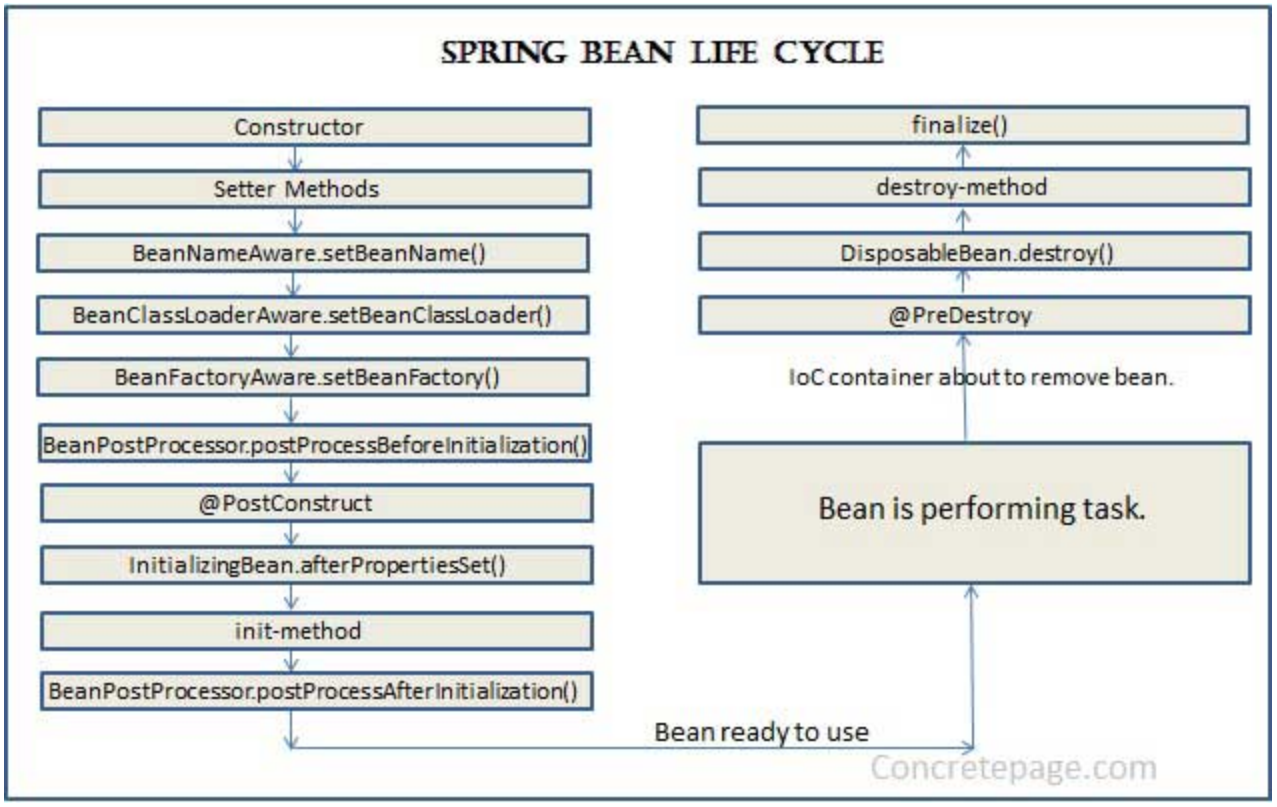
Bean lifecycle methods



- You can add custom code during bean initialization
    - Calling custom business logic methods
    - Setting up handles to resources (db, sockets, file etc)
- You can add custom code during bean destruction
    - Calling custom business logic method
    - Clean up handles to resources (db, sockets, files etc)

Spring bean lifecycle

Reference: https://www.concretepage.com/spring/spring-bean-life-cycle-tutorial



Specifying bean init and destroy methods in xml configuration

```
<bean id="myCoach"
                    class="com.luv2code.springdemo.TrackCoach"
                    scope="singleton"
                    init-method="doMyStartupStuff"
                    destroy-method="doMyCleanupStuffYoYo">
```

Spring configuration with annotations

- XML configuration can be verbose
- Configure your Spring beans with Annotations
- Annotations minimizes the XML configuration

What are Java Annotations?

- Special labels/markers added to Java classes
- Provide meta-data about the class
- Processed at compile time or run-time for special processing

Component scanning

Spring will scan your Java classes for special annotations and automatically register the beans in the Spring container
Steps to achieve component scanning
1. Enable component scanning in Spring config file
2. Add the @Component Annotation to your Java classes
3. Retrieve bean from Spring container

```
<beans … >
   <context:component-scan base-package="com.luv2code.springdemo" />
</beans>
```

```
@Component("thatSillyCoach")
public class TennisCoach implements Coach {
   @Override
   public String getDailyWorkout() {
       return "Practice your backhand volley";
   }
}
```

```
Coach theCoach = context.getBean("thatSillyCoach", Coach.class);
```

Specifying bean scope using annotation

```
@Component
@Scope("prototype")
public class TennisCoach implements Coach {
//...
}
```

Bean Id

We can specify bean id in the component annotation

```
@Component("thatSillyCoach")
```

If bean id is not specified in the annotation, then bean id is the class name with first letter lowercase

```
@Component
public class TennisCoach implements Coach {
```

Bean declared above will have bean id as “tennisCoach”

Autowiring

- For dependency injection, Spring can use auto wiring.
- Spring will look for a class that matches the property (matches by type: class or interface). Spring will scan @Components
- Spring will inject it automatically, hence it is Autowired
- Autowiring can significantly reduce the need to specify properties or constructor arguments.
- Autowiring can update a configuration as your objects evolve.
- Autowiring injection can be done for the following
    - Constructor Injection
    - Setter Injection (any method with parameters)
    - Field Injections

```
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;

@Component
```

```
public class TennisCoach implements Coach {
    // field injection
    //@Autowired
    private FortuneService fortuneService;

    // define a default constructor
    public TennisCoach() {
        System.out.println(">> TennisCoach: inside default constructor");
    }

    // define a setter method
    @Autowired
    public void setFortuneService(FortuneService theFortuneService) {
        System.out.println(">> TennisCoach: inside setFortuneService() method");
        this.fortuneService = theFortuneService;
    }

    // constructor injection
    /* @Autowired
    public TennisCoach(FortuneService theFortuneService) {
        fortuneService = theFortuneService;
    }
    */
    @Override
    public String getDailyWorkout() {
        return "Practice your backhand volley";
    }
    @Override
    public String getDailyFortune() {
        return fortuneService.getFortune();
    }
}
```

Qualifier

If we have multiple implementations of an interface and if we try to Autowire, then spring will throw NoUniqueBeanException as it is not clear which implementation to inject. In this case we can use “@Qualifier” annotation to specify the bean id to be injected

```
@Component
public class TennisCoach implements Coach {
    @Autowired
    @Qualifier("happyFortuneService")
    private FortuneService fortuneService;
    //..
}
```

Marking primary bean

```
/*
When declaring beans, you can avoid autowiring ambiguity by designating one of
the candidate beans as a primary bean. In the event of any ambiguity, Spring will
choose the primary bean over any other candidate beans
*/

@Component
@Primary
public class IceCream implements Dessert { ... }

@Bean
@Primary
public Dessert iceCream() {
return new IceCream();
}
```

Specifying bean init and destroy methods using annotations

```
@Component
public class TennisCoach implements Coach {

   @Autowired
   @Qualifier("randomFortuneService")
   private FortuneService fortuneService;

   // define a default constructor
   public TennisCoach() {
       System.out.println(">> TennisCoach: inside default constructor");
   }

   // define my init method
   @PostConstruct
   public void doMyStartupStuff() {
       System.out.println(">> TennisCoach: inside of doMyStartupStuff()");
   }

   // define my destroy method
   @PreDestroy
   public void doMyCleanupStuff() {
       System.out.println(">> TennisCoach: inside of doMyCleanupStuff()");
   }
}
```

Annotation based java configuration of beans

```
@Configuration
```

```
@ComponentScan //will automatically scan same package in which configuration class is present for
components
public class CDPlayerConfig {}
```

```
//Explicitly specify basepackage for component scan
@Configuration
@ComponentScan("soundsystem")
public class CDPlayerConfig {}
```

```
//specify basePackageClasses for component scan
//Whatever packages those classes are in will be used as the base package for component scanning
//consider creating an empty marker interface in the packages to be scanned. With a
//marker interface, you can still have a refactor-friendly reference to an interface, but
//without references to any actual application code
@Configuration
@ComponentScan(basePackageClasses={CDPlayer.class, DVDPlayer.class})
public class CDPlayerConfig {}
```

```
//Explicit bean construction using java code
import org.springframework.context.annotation.Configuration;

@Configuration
public class CDPlayerConfig {
   @Bean
   public CompactDisc sgtPeppers() {
           return new SgtPeppers();
   }
}
```

Injecting property values using @value annotation

```
# datasource.properties
guru.username=superuser
guru.password=dbPassword
guru.jdbcurl=someUrlforDB
```

```
//injecting properties using @Value annotation
@PropertySource("classpath:datasource.properties")
@ImportResource("classpath:sfgdi-config.xml")
@Configuration
public class GreetingServiceConfig {

   @Bean
   FakeDataSource fakeDataSource(@Value("${guru.username}") String username,
                                 @Value("${guru.password}") String password,
                                 @Value("${guru.jdbcurl}") String jdbcurl){
       FakeDataSource fakeDataSource = new FakeDataSource();
       fakeDataSource.setUsername(username);
       fakeDataSource.setPassword(password);
       fakeDataSource.setJdbcurl(jdbcurl);
       return fakeDataSource;
   }
```

```
}
```

Lazy initialization of beans

When we put @Lazy annotation over the @Configuration class, it indicates that all the methods with @Bean annotation should be loaded lazily. For componentScan annotation, put lazyInit = true, to load beans lazily

This is the equivalent for the XML based configuration's default-lazy-init="true" attribute.
References:
[How to load all beans lazily with @ComponentScan in Spring?]
https://www.baeldung.com/spring-lazy-annotation

```
@Lazy
@Configuration
@ComponentScan(basePackages = "com.baeldung.lazy", lazyInit = true)
public class AppConfig {

   @Bean
   public Region getRegion(){
       return new Region();
   }

   @Bean
   public Country getCountry(){
       return new Country();
   }
}
```

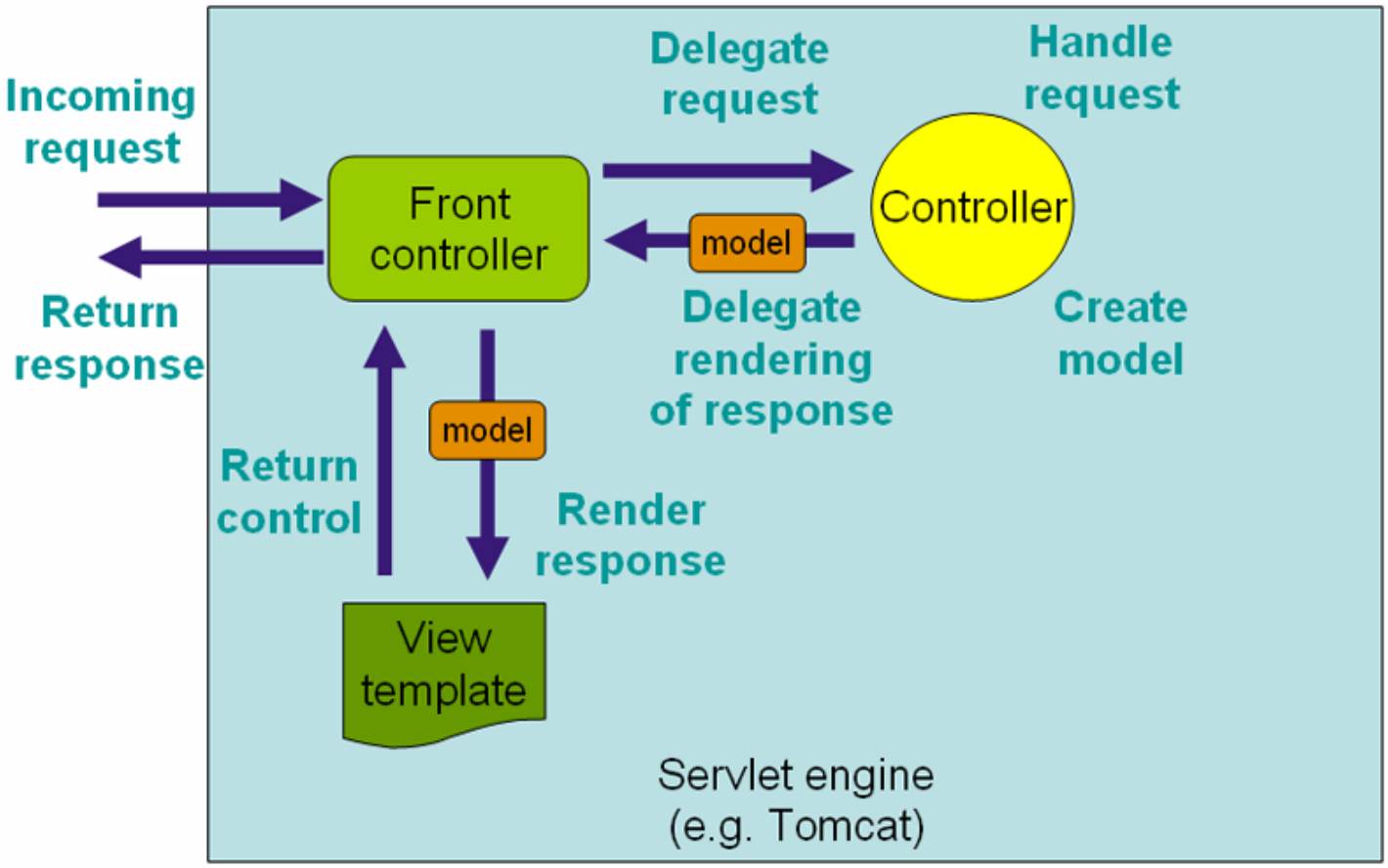
Activating bean based on profile

```
import javax.activation.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
@Configuration
@Profile("dev")
public class DevelopmentProfileConfig {
       @Bean(destroyMethod="shutdown")
       public DataSource dataSource() {
               return new EmbeddedDatabaseBuilder()
               .setType(EmbeddedDatabaseType.H2)
               .addScript("classpath:schema.sql")
               .addScript("classpath:test-data.sql")
               .build();
       }
}
```

Spring MVC

Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning. Spring MVC, as many other web frameworks, is designed around the front controller pattern where a central `Servlet`, the `DispatcherServlet`, provides a shared algorithm for request processing, while actual work is performed by configurable delegate components.

- Based on Model-View-Controller design pattern
- Leverages features of the Core Spring Framework (IoC, DI)



Components of a Spring MVC Application

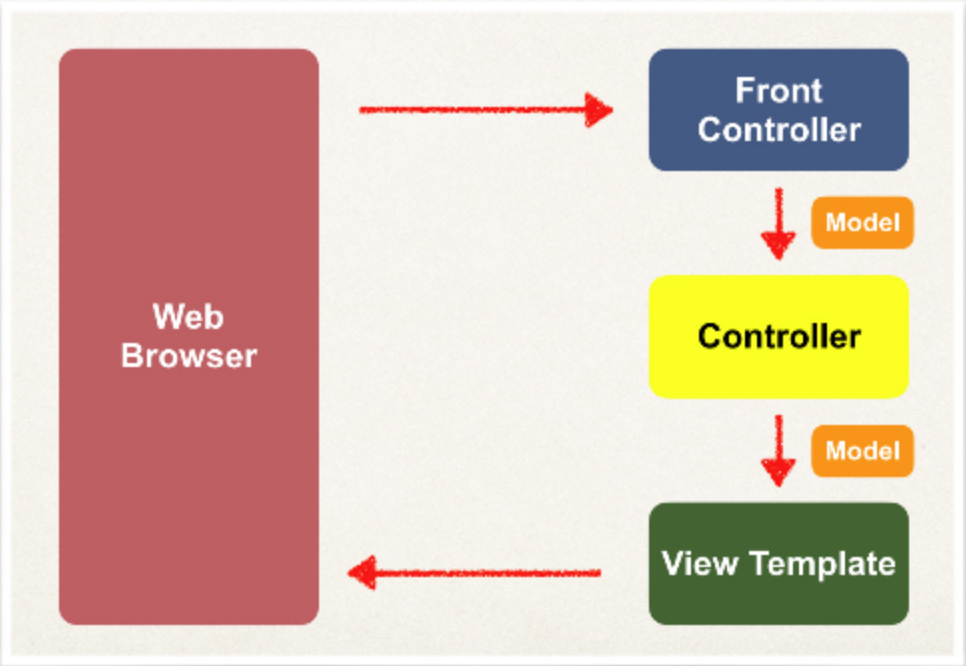
- A set of web pages to layout UI components
- A collection of Spring beans (controllers, services, etc...)
- Spring configuration (XML, Annotations or Java)

Spring MVC Front Controller

- Front controller known as DispatcherServlet
- Part of the Spring Framework
- Already developed by Spring Dev Team

We have to develop

- Model objects (orange in below picture)
- View templates (dark green)
- Controller classes (yellow)



Controller

- Contains your business logic
- Handle the request
- Store/retrieve data (db, web service…)
- Place data in model
- Route to appropriate view template

Model

- Model - contains your data
- Store/retrieve data via backend systems
    - database, web service, etc…
    - Use a Spring bean if you like
- Place your data in the model
    - Data can be any Java object/collection

View Template

- Spring MVC is flexible and it supports many rendering frameworks for view templates
    - Most common is JSP + JSTL
    - Supports thymeleaf, groovy pages, freemarker etc
- Developer has to create a page which displays the data

Spring MVC lifecycle of a request

Reference : Spring in Action 4th edition by craig walls

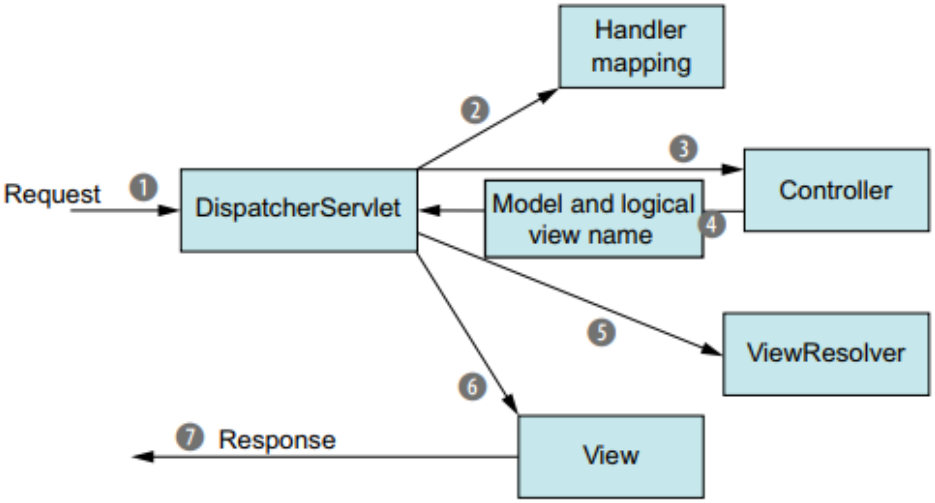
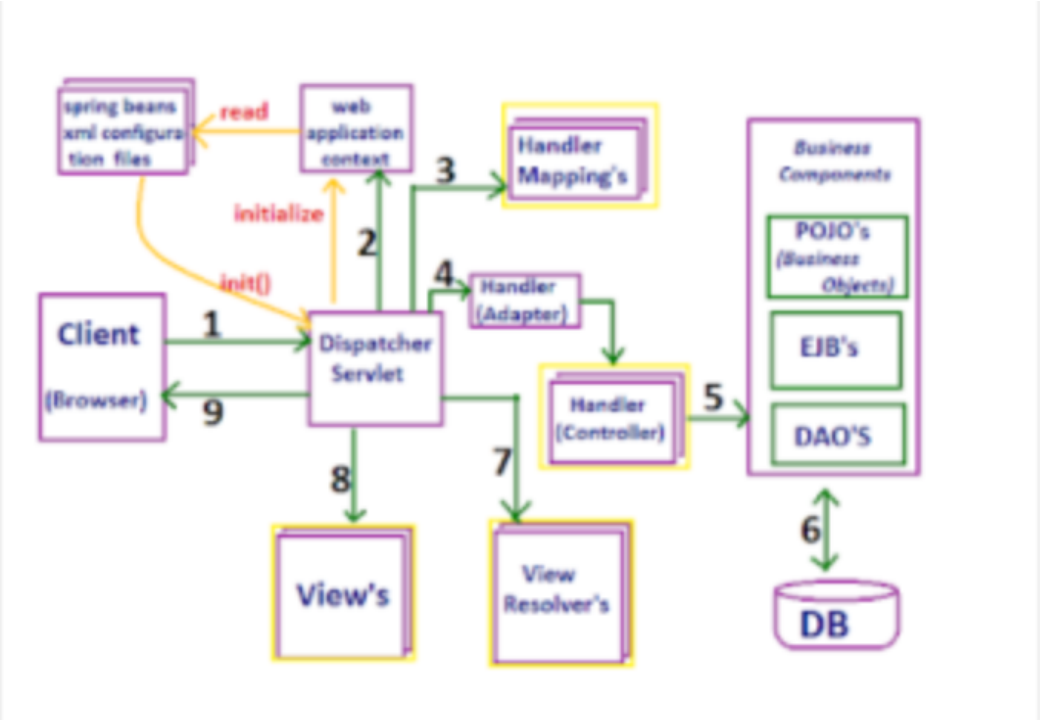
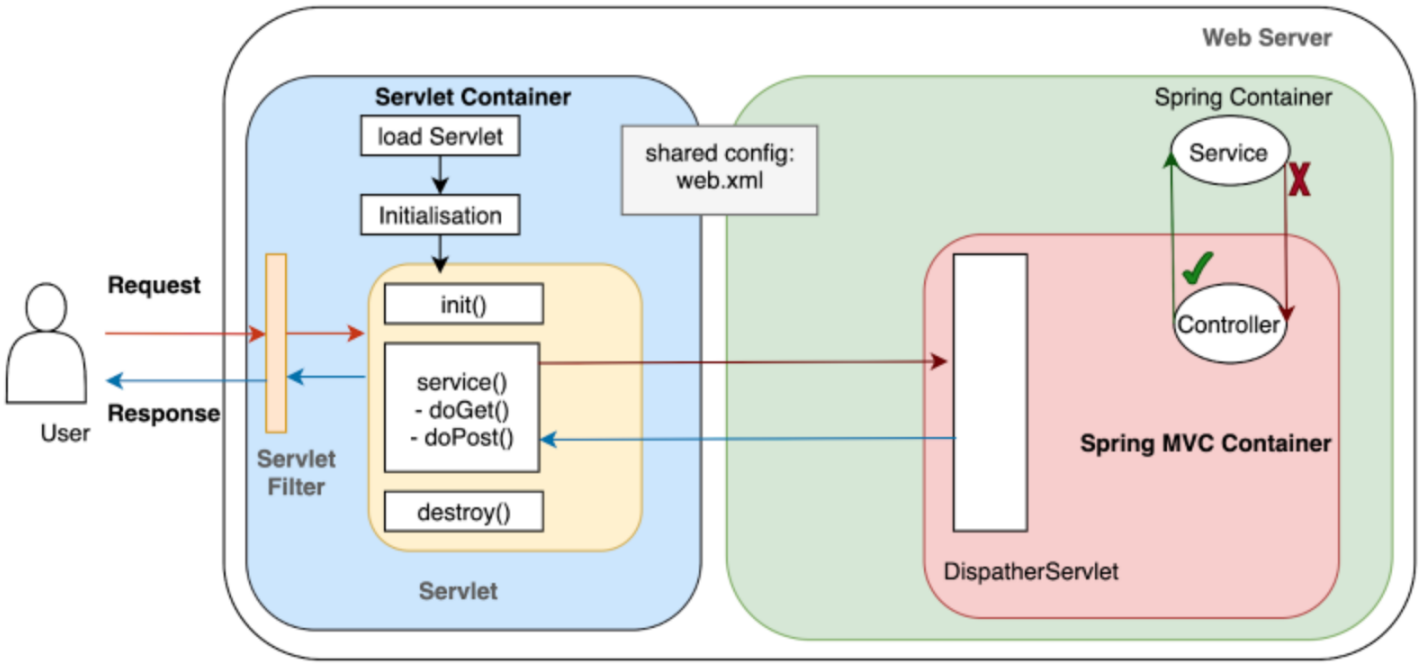


Figure 5.1 A request couriers information to several stops on its way to producing the desired results.

1. Request leaves the browser, it carries information about what the user is asking for. At the least, the request will the requested URL.
2. The DispatcherServlet's job is to send the request on to a Spring MVC controller. A controller is a Spring component that processes the request. But a typical application may have several controllers, So the DispatcherServlet consults one or more handler mappings (they map URL pattern to controllers) to figure out where the request's next stop will be.
3. DispatcherServlet sends the request to the chosen controller. At the controller, the request parameters and payload are examined and processing is done
    a. The logic performed by a controller often results in some information that needs to be carried back to the user and displayed in the browser. This information is referred to as the model. But sending raw information back to the user isn't sufficient—it needs to be formatted in a user-friendly format, typically HTML. For that, the information needs to be given to a view, typically a JavaServer Page (JSP).
4. Controller sends the request, along with the model and view name, back to the DispatcherServlet
    a. The view name passed back to DispatcherServlet doesn't directly identify a specific JSP. It doesn't even necessarily suggest that the view is a JSP. Instead, it only carries a logical name that will be used to look up the actual view that will produce the result. This is so that the controller doesn't get coupled to a particular view and that routing logic is independent of presentation technology
5. DispatcherServlet consults a view resolver to map the logical view name to a specific view implementation, which may or may not be a JSP.
6. Request's final stop is at the view implementation, typically a JSP, where it delivers the model data. The request's job is finally done.
7. The view will use the model data to render output that will be carried back to the client by the response object





How dispatcher servlet is scanned by spring

In a Servlet 3.0 environment, the container looks for any classes in the classpath that implement the javax.servlet.ServletContainerInitializer interface; if any are found, they're used to configure the servlet container.

Spring supplies an implementation of that interface called SpringServletContainerInitializer that, in turn, seeks out any classes that implement WebApplicationInitializer and delegates to them for configuration. Spring 3.2 introduced a convenient base implementation of WebApplicationInitializer called AbstractAnnotationConfigDispatcherServletInitializer. Because your SpittrWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer (and thus implements WebApplicationInitializer), it will be automatically discovered when deployed in a Servlet 3.0 container and be used to configure the servlet context

Configuring dispatcher servlet in java

We can also configure more than one dispatcher servlet,configure servlet using web.xml, have rootconfig and webconfig etc.
Refer Spring Framework file for details

```
import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
public class SpittrWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
   @Override
   protected String[] getServletMappings() {
       return new String[] {"/"};
   }

   @Override
   protected Class<?>[] getRootConfigClasses() {
       return new Class<?>[] {RootConfig.class};
   }

   @Override
   protected Class<?>[] getServletConfigClasses() {
       return new Class<?>[] {WebConfig.class};
   }
}
```

When DispatcherServlet starts up, it creates a Spring application context and starts loading it with beans declared in the configuration files or classes that it's given. With the getServletConfigClasses() method you've asked that DispatcherServlet load its application context with beans defined in the WebConfig configuration class (using Java configuration).

But in Spring web applications, there's often another application context. This other application context is created by ContextLoaderListener.

DispatcherServlet is expected to load beans containing web components such as controllers, view resolvers, and handler mappings. ContextLoaderListener is expected to load the other beans in your application. These beans are typically the middle-tier and data-tier components that drive the back end of the application.These beans are shared by multiple DispatcherServlet like beans

Under the covers, AbstractAnnotationConfigDispatcherServletInitializer creates both a DispatcherServlet and a ContextLoaderListener. The @Configuration classes returned from getServletConfigClasses() will define beans for DispatcherServlet's application context. Meanwhile, the @Configuration class's returned getRootConfigClasses() will be used to configure the application context created by ContextLoaderListener.

There can be multiple dispatcher servlets in spring mvc application. Each dispatcher servlet will operate in its own namespace, loading its own application context (that we specify) with mappings, handlers, etc.

ContextLoaderListener creates the root application context and will be shared with child contexts created by all DispatcherServlet contexts. There can be only one root application context in spring mvc application. The

context of ContextLoaderListener contains beans that globally visible, like services, repositories, infrastructure beans, etc.

Configuring web config for spring mvc

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc // Enable spring mvc
@ComponentScan("spitter.web") // Enable component scanning
public class WebConfig extends WebMvcConfigurerAdapter {
   // Configure a JSP view resolver
   @Bean
   public ViewResolver viewResolver() {
       InternalResourceViewResolver resolver = new InternalResourceViewResolver();
       resolver.setPrefix("/WEB-INF/views/");
       resolver.setSuffix(".jsp");
       resolver.setExposeContextBeansAsAttributes(true);
       return resolver;
   }

   // Configure static content handling
   @Override
   public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
       configurer.enable();
   }
}
```

Configuring root config

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.ComponentScan.Filter;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
@Configuration
@ComponentScan(basePackages={"spitter"},
excludeFilters={
@Filter(type=FilterType.ANNOTATION, value=EnableWebMvc.class)
})
public class RootConfig {
```

```
}
```

Writing controllers

Sample index controller

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class IndexController {

   @RequestMapping({"", "/", "index", "index.html"})
   public String index(){

       return "index";
   }
}
```

Controller for routing to specific views

```
@Controller
public class HelloWorldController {
// need a controller method to show the initial HTML form
@RequestMapping("/showForm")
public String showForm() {
return "helloworld-form";
}
// need a controller method to process the HTML form
@RequestMapping("/processForm")
public String processForm() {
return "helloworld";
}
}
```

Passing model to controller and getting request param

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(HttpServletRequest request, Model model) {
   // read the request parameter from the HTML form
   String theName = request.getParameter("studentName");
   // convert the data to all caps
   theName = theName.toUpperCase();
   // create the message
   String result = "Yo! " + theName;
   // add message to the model
   model.addAttribute("message", result);
   return "helloworld";
}
```

View template - JSP

```
<html>
   <body>
   Hello World of Spring!
   ...
   The message: ${message}
   </body>
</html>
```

Controller with request parameter injected also having default value

```
private static final String MAX_LONG_AS_STRING =Long.toString(Long.MAX_VALUE);

@RequestMapping(method = RequestMethod.GET)
public List<Spittle> spittles(
       @RequestParam(value = "max", defaultValue = MAX_LONG_AS_STRING) long max,
       @RequestParam(value = "count", defaultValue = "20") int count) {
   return spittleRepository.findSpittles(max, count);
}
```

Controller with path parameter

```
@RequestMapping(value = "/{spittleId}", method = RequestMethod.GET)
public String spittle(@PathVariable("spittleId") long spittleId, Model model) {
   model.addAttribute(spittleRepository.findOne(spittleId));
   return "spittle";
}

// If no value attribute is given for @PathVariable, it assumes the placeholder's name is
// the same as the method parameter name.
@RequestMapping(value = "/{spittleId}", method = RequestMethod.GET)
public String spittle(@PathVariable long spittleId, Model model) {
   model.addAttribute(spittleRepository.findOne(spittleId));
   return "spittle";
}
```

Controller handling POST method with params

```
@RequestMapping(value = "/register", method = POST)
public String processRegistration(Spitter spitter) {
   spitterRepository.save(spitter);                                    // save a spitter
   return "redirect:/spitter/" + spitter.getUsername();          //redirect to profile page
}
```

Spring MVC form tags

- Spring MVC Form Tags are the building block for a web page
- Form Tags are configurable and reusable for a web page
- Spring MVC Form Tags can make use of data binding
- Automatically setting / retrieving data from a Java object / bean

```java
//Student.java

package com.luv2code.springdemo.mvc;
import java.util.LinkedHashMap;
public class Student {
   private String firstName;
   private String lastName;
   private String country;
   private LinkedHashMap<String, String> countryOptions;
   private String favoriteLanguage;
   private String[] operatingSystems;
   public Student() {
           // populate country options: used ISO country code
           countryOptions = new LinkedHashMap<>();
           countryOptions.put("BR", "Brazil");
           countryOptions.put("FR", "France");
           countryOptions.put("DE", "Germany");
           countryOptions.put("IN", "India");
           countryOptions.put("US", "United States of America");
   }
   public String getFirstName() {
           return firstName;
   }
   public void setFirstName(String firstName) {
           this.firstName = firstName;
   }
   public String getLastName() {
           return lastName;
   }
   public void setLastName(String lastName) {
           this.lastName = lastName;
   }
   public String getCountry() {
           return country;
   }
   public void setCountry(String country) {
           this.country = country;
   }
   public LinkedHashMap<String, String> getCountryOptions() {
           return countryOptions;
   }
   public String getFavoriteLanguage() {
           return favoriteLanguage;
   }
   public void setFavoriteLanguage(String favoriteLanguage) {
           this.favoriteLanguage = favoriteLanguage;
   }
```

```
   public String[] getOperatingSystems() {
           return operatingSystems;
   }
   public void setOperatingSystems(String[] operatingSystems) {
           this.operatingSystems = operatingSystems;
   }
}
```

```
//StudentController.java

package com.luv2code.springdemo.mvc;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@RequestMapping("/student")
public class StudentController {
   @RequestMapping("/showForm")
   public String showForm(Model theModel) {
           // create a student object
           Student theStudent = new Student();
           // add student object to the model
           theModel.addAttribute("student", theStudent);
           return "student-form";
   }
   @RequestMapping("/processForm")
   public String processForm(@ModelAttribute("student") Student theStudent) {
           // log the input data
           System.out.println("theStudent: " + theStudent.getFirstName()
                                                   + " " + theStudent.getLastName());
           return "student-confirmation";
   }
}
```

```
<!-- student-form.jsp -->

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<!DOCTYPE html>
<html>
<head>
       <title>Student Registration Form</title>
</head>
<body>
       <form:form action="processForm" modelAttribute="student">
               First name: <form:input path="firstName" />
               <br><br>
```

```
		Last name: <form:input path="lastName" />
		<br><br>
		Country:
		<form:select path="country">
			<form:options items="${student.countryOptions}" />
		</form:select>
		<br><br>
		Favorite Language:
		Java <form:radiobutton path="favoriteLanguage" value="Java" />
		C# <form:radiobutton path="favoriteLanguage" value="C#" />
		PHP <form:radiobutton path="favoriteLanguage" value="PHP" />
		Ruby <form:radiobutton path="favoriteLanguage" value="Ruby" />
		<br><br>
		Operating Systems:
		Linux <form:checkbox path="operatingSystems" value="Linux" />
		Mac OS <form:checkbox path="operatingSystems" value="Mac OS" />
		MS Windows <form:checkbox path="operatingSystems" value="MS Window" />
		<br><br>
		<input type="submit" value="Submit" />
	</form:form>

</body>
</html>
```

```
<!-- student-confirmation.jsp -->

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
<head>
	<title>Student Confirmation</title>
</head>
<body>
The student is confirmed: ${student.firstName} ${student.lastName}
<br><br>
Country: ${student.country}
<br><br>
Favorite Language: ${student.favoriteLanguage}
<br><br>
Operating Systems:
<ul>
	<c:forEach var="temp" items="${student.operatingSystems}">
		<li> ${temp} </li>
	</c:forEach>
</ul>
</body>
</html>
```

Tag list

```
<sf:checkbox>               Renders an HTML <input> tag with type set to checkbox.
<sf:checkboxes>             Renders multiple HTML <input> tags with type set to checkbox.
<sf:errors>                 Renders field errors in an HTML <span> tag.
<sf:form>                   Renders an HTML <form> tag and exposed binding path to inner tags for
data-binding.

<sf:hidden>                 Renders an HTML <input> tag with type set to hidden.
<sf:input>                  Renders an HTML <input> tag with type set to text.
<sf:label>                  Renders an HTML <label> tag.
<sf:option>                 Renders an HTML <option> tag. The selected attribute is set according
to the bound value.

<sf:options>                Renders a list of HTML <option> tags corresponding to the
boundcollection, array, or map.

<sf:password>               Renders an HTML <input> tag with type set to password.
<sf:radiobutton>            Renders an HTML <input> tag with type set to radio.
<sf:radiobuttons>           Renders multiple HTML <input> tags with type set to radio.
<sf:select>                 Renders an HTML <select> tag.
<sf:textarea>               Renders an HTML <textarea> tag
```

Bean validation API and Spring

- Java has a standard Bean Validation API
- Defines a metadata model and API for entity validation
- Not tied to either the web tier or the persistence tier
- Available for server-side apps and also client-side JavaFX/Swing apps
- Reference: http://www.beanvalidation.org
- Spring version 4 and higher supports Bean Validation API
- Preferred method for validation when building Spring apps
- To use, simply add validation JARs to our project

Bean validation annotations

Annotation	Description
@AssertFalse	The annotated element must be a Boolean type and be false.
@AssertTrue	The annotated element must be a Boolean type and be true.
@DecimalMax	The annotated element must be a number whose value is less than or equal to a given BigDecimalString value.
@DecimalMin	The annotated element must be a number whose value is greater than or equal to a given BigDecimalString value.
@Digits	The annotated element must be a number whose value has a specified number of digits.

@Future	The value of the annotated element must be a date in the future.
@Max	The annotated element must be a number whose value is less than or equal to a given value.
@Min	The annotated element must be a number whose value is greater than or equal to a given value.
@NotNull	The value of the annotated element must not be null.
@Null	The value of the annotated element must be null.
@Past	The value of the annotated element must be a date in the past.
@Pattern	The value of the annotated element must match a given regular expression.
@Size	The value of the annotated element must be either a String, a collection, or an array whose length fits within the given range

Bean validation implementation

```
<!-- Hibernate Validator -->
<dependency>
   <groupId>org.hibernate.validator</groupId>
   <artifactId>hibernate-validator</artifactId>
   <version>7.0.1.Final</version>
</dependency>
```

Bean validation example code

```
//Customer.java POJO with validation annotations
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

public class Customer {
   private String firstName;

   @NotNull(message = "is required")
   @Size(min = 1, message = "is required")
   private String lastName;

   @NotNull(message = "is required")
   @Min(value = 0, message = "must be greater than or equal to zero")
   @Max(value = 10, message = "must be less than or equal to 10")
   private Integer freePasses;

   @Pattern(regexp = "^[a-zA-Z0-9]{5}", message = "only 5 chars/digits")
   private String postalCode;
```

```
    //setters and getters
}
```

```
//Customer controller
import javax.validation.Valid;
import org.springframework.beans.propertyeditors.StringTrimmerEditor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/customer")
public class CustomerController {

    // add an initbinder ... to convert trim input strings
    // remove leading and trailing whitespace
    // resolve issue for our validation
    @InitBinder
    public void initBinder(WebDataBinder dataBinder) {
        StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);
        dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);
    }


    @RequestMapping("/showForm")
    public String showForm(Model theModel) {
        theModel.addAttribute("customer", new Customer());
        return "customer-form";
    }

    // the BindingResult parameter must appear immediately after the model attribute.
    // If you place it in any other location, Spring MVC validation will not work as desired. In
fact, your validation rules will be ignored.
    @RequestMapping("/processForm")
    public String processForm(@Valid @ModelAttribute("customer") Customer theCustomer,
            BindingResult theBindingResult) {

        System.out.println("Last name: |" + theCustomer.getLastName() + "|");

        if (theBindingResult.hasErrors()) {
            return "customer-form";
        } else {
            return "customer-confirmation";
```

```
        }
    }
}
```

```
<!-- customer-form.jsp -->

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
    <title>Customer Registration Form</title>
    <style>
        .error {
            color: red
        }
    </style>
</head>
<body>
    <i>Fill out the form. Asterisk (*) means required.</i>
    <br><br>
    <form:form action="processForm" modelAttribute="customer">
        First name:
        <form:input path="firstName" />
        <br><br>
        Last name (*):
        <form:input path="lastName" />
        <form:errors path="lastName" cssClass="error" />
        <br><br>
        Free passes (*):
        <form:input path="freePasses" />
        <form:errors path="freePasses" cssClass="error" />
        <br><br>
        Postal Code:
        <form:input path="postalCode" />
        <form:errors path="postalCode" cssClass="error" />
        <br><br>
        <input type="submit" value="Submit" />
    </form:form>
</body>
</html>
```

```
<!-- customer-confirmation.jsp -->

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
<head>
    <title>Customer Confirmation</title>
```

```
</head>
<body>
   The customer is confirmed: ${customer.firstName} ${customer.lastName}
   <br><br>
   Free passes: ${customer.freePasses}
   <br><br>
   Postal Code: ${customer.postalCode}
</body>
</html>
```

Spring security

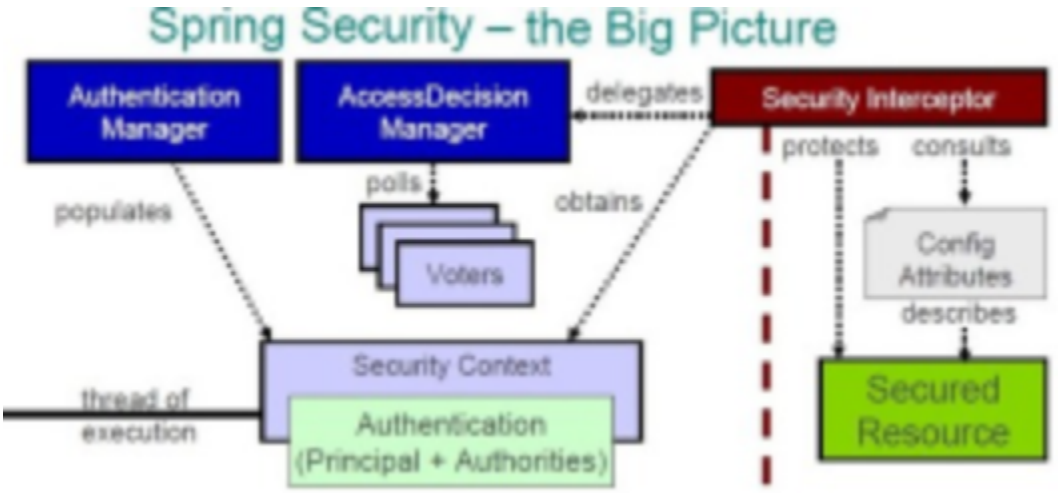
Spring Security is a security framework that provides declarative security for your Spring-based applications. Spring Security provides a comprehensive security solution, handling authentication and authorization at both the web request level and at the method invocation level.

Spring Security employs several servlet filters to provide various aspects of security, you'll only need to configure one of those filters. DelegatingFilterProxy is a special servlet filter that, by itself, doesn't do much. Instead, it delegates to an implementation of javax.servlet.Filter that's registered as a <bean> in the Spring application context.

The bean is named "springSecurityFilterChain", which is an internal infrastructure bean created by the namespace to handle web security. springSecurityFilterChain bean itself is another special filter known as FilterChainProxy. It's a single filter that chains together one or more additional filters. Spring Security relies on several servlet filters to provide different security features, but you should almost never need to know these details, as you likely won't need to explicitly declare the springSecurityFilterChain bean or any of the filters it chains together. Those filters will be created when you enable web security.

Spring security supports In-memory, JDBC, LDAP, Custom / Pluggable authentication data stores

Spring security big picture



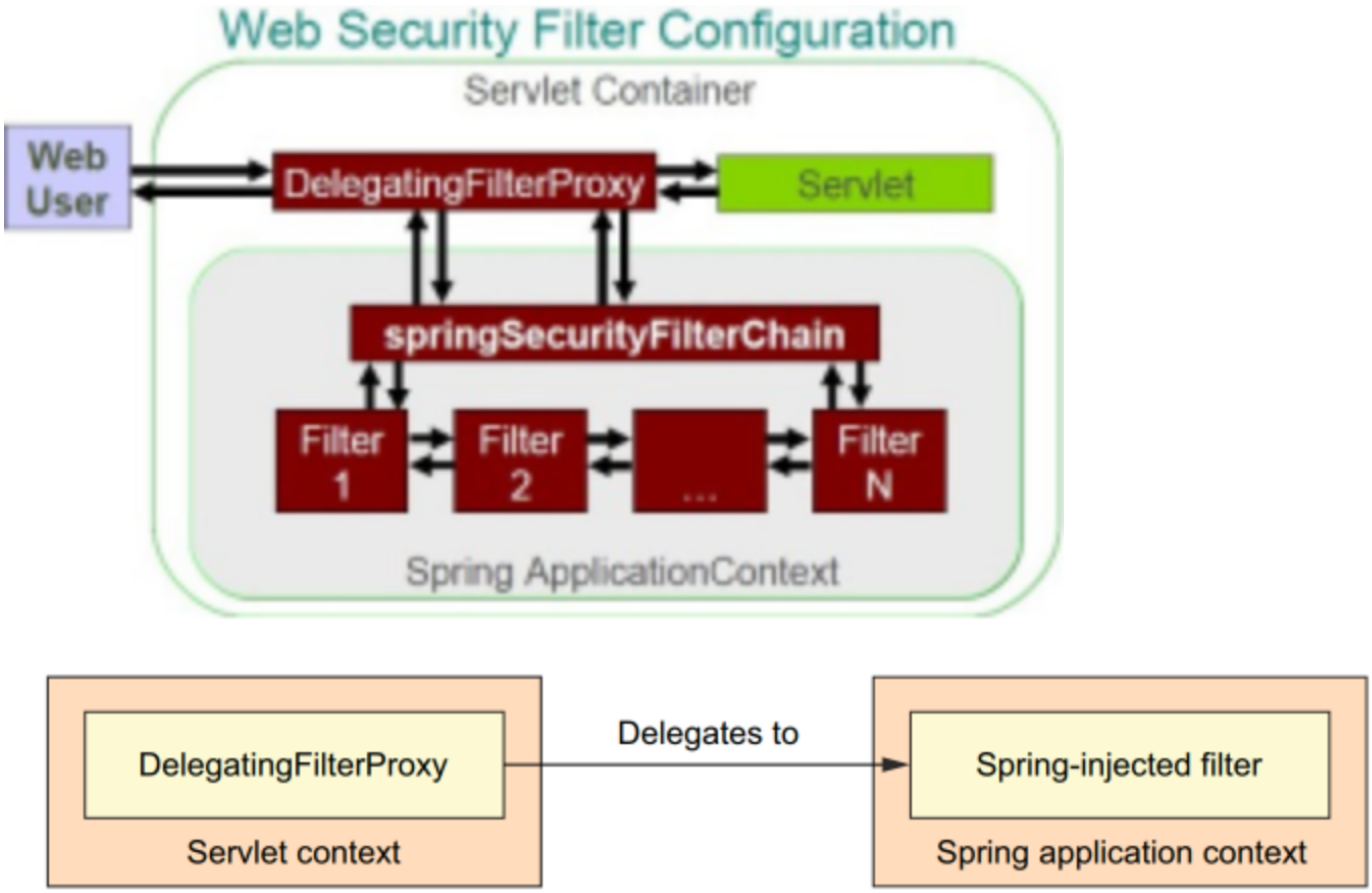


Figure 9.1 `DelegatingFilterProxy` proxies filter handling to a delegate filter bean in the Spring application context.

Spring security @EnableWebSecurity annotation

The @EnableWebSecurity annotation enables web security. It is useless on its own, however. Spring Security must be configured in a bean that implements WebSecurityConfigurer or (for convenience) extends WebSecurityConfigurerAdapter. Any bean in the Spring application context that implements WebSecurityConfigurer can contribute to Spring Security configuration, but it's often most convenient for the configuration class to extend WebSecurityConfigurerAdapter

@EnableWebSecurity is generally useful for enabling security in any web application. But if you happen to be developing a Spring MVC application, you should consider using @EnableWebMvcSecurity instead. the @EnableWebMvcSecurity annotation configures a Spring MVC argument resolver so that handler methods can receive the authenticated user's principal (or username) via @AuthenticationPrincipal-annotated parameters. It also configures a bean that automatically adds a hidden cross-site request forgery (CSRF) token field on forms using Spring's form-binding tag library

Method	**Description**
configure(WebSecurity)	Override to configure Spring Security's filter chain.
configure(HttpSecurity)	Override to configure how requests are secured by interceptors.(configure security of web paths in application,

	login, logout etc)
configure(AuthenticationManagerBuilder)	Override to configure user-details services(in memory, database, ldap, etc)

```
@Configuration
@EnableWebMvcSecurity
public class DemoSecurityConfig extends WebSecurityConfigurerAdapter {
	// add a reference to our security data source
	@Autowired
	private DataSource securityDataSource;

	@Override
	protected void configure(AuthenticationManagerBuilder auth) throws Exception {
		// use jdbc authentication
		auth.jdbcAuthentication().dataSource(securityDataSource);
	}

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

		http.authorizeRequests()
			.antMatchers("/").hasRole("EMPLOYEE")
			.antMatchers("/leaders/**").hasRole("MANAGER")
			.antMatchers("/systems/**").hasRole("ADMIN")
			.and()
			.formLogin()
				.loginPage("/showMyLoginPage")
				.loginProcessingUrl("/authenticateTheUser")
				.permitAll()
			.and()
			.logout().permitAll()
			.and()
			.exceptionHandling().accessDeniedPage("/access-denied");
	}
}
```

Spring security login, logout, csrf, access denied configuration in java and xml

```
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder
;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
```

```
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.User.UserBuilder;

@Configuration
@EnableWebSecurity
public class DemoSecurityConfig extends WebSecurityConfigurerAdapter {

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

       http
           // any request to the app must be authenticated
           .authorizeRequests()
               .anyRequest().authenticated()
           .and()
               // we are customizing form login process
               .formLogin()
               // show our custom form at request mapping "/showMyLoginPage" as specified below
               .loginPage("/showMyLoginPage")
               // login form should POST data to this URL for processing
               .loginProcessingUrl("/authenticateTheUser")
               // any user can login
               .permitAll()
               // use .defaultSuccessUrl method and .failureUrl method to have custom success and
error login URLs
           // need to create controller code and custom access denied page
           .and()
               .exceptionHandling().accessDeniedPage("/access-denied")
               // use below code to return http code access denied
               /*.exceptionHandling()
                   .accessDeniedPage("/403")*/
           and()
               // while adding logout button, send data to default logout URL: /logout, must use
POST method
               .logout() // configure logout
               .permitAll(); // any user can access
               // when logout is processed, spring security will invalidate user's http session and
remove session cookies, etc
               // it will also send user back to login page and append a logout parameter ?logout
               // in login page check, if logout parameter exists show "logged out" message
               // use logout().logoutSuccessUrl method to specify custom logout URL
           // use below code to disable csrf
           .csrf().disable()
           // if enabling cross site request forgery (CSRF) protection, then
           // for form submissions use POST instead of GET
           // Include csrf token in form submissions
           //        springs <form:form> automatically adds CSRF token
```

```
            //        if you don't use <form:form>, you must manually add CSRF form token
            /*
            <input type="hidden"
                                name="${_csrf.parameterName}"
                                value="${_csrf.token}" />
            */
            ;
    }
}
```

Spring security ant matcher methods and other http methods

```
// antMatcher conditions are evaluated against a given URL in the sequence they are specified in
the below builder after call to authorizeRequests method
http
.authorizeRequests()
// To access URL "/spitters/me" user should be logged in
.antMatchers("/spitters/me").authenticated()
// To do http POST (create/modify data) user should be authenticated
.antMatchers(HttpMethod.POST, "/spittles").authenticated()
// Permit all other URLS with no need for authentication or authorities
.anyRequest().permitAll();

// match any URL starting with "/spitters" by using ** wildcard
.antMatchers("/spitters/**").authenticated();

// specify multiple path/URL
.antMatchers("/spitters/**", "/spittles/mine").authenticated();

// match URL path's with regular expression
.regexMatchers("/spitters/.*").authenticated();

// allow URL path to be accessed by users having specific role
.antMatchers("/spitters/me").hasAuthority("ROLE_SPITTER")
// use hasRole method to specify role without prefix (default value for role prefix is "ROLE_")
.antMatchers("/spitter/me").hasRole("SPITTER")

// using SpEL expression for authentication
.antMatchers("/spitter/me")
.access("hasRole('ROLE_SPITTER') and hasIpAddress('192.168.1.2')")

// enforcing channel security (https)
// first you can specify any antmatchers for URL path which do not require https
// and then use and() method to specify URL's requring https
.and()
.requiresChannel()
.antMatchers("/spitter/form").requiresSecure();
```

```
// declare that home page be sent always over http
.antMatchers("/").requiresInsecure();

// enable http basic authentication, for REST controllers
http
.formLogin()
.loginPage("/login")
.and()
.httpBasic()
.realmName("Spittr")
.and()
...

// enable remember me functionality to avoid forcing users to login everytime
http
.formLogin()
.loginPage("/login")
.and()
.rememberMe()
.tokenValiditySeconds(2419200)
.key("spittrKey")
```

You can chain as many calls to antMatchers(), regexMatchers(), and anyRequest() as you need to fully establish the security rules around your web application. You should know, however, that they'll be applied in the order given. For that reason, it's important to configure the most specific request path patterns first and the least specific ones (such as anyRequest()) last. If not, then the least specific paths will trump the more specific ones.
Note: antMatcher method was deprecated in spring 6, so use requestMatcher method instead

Configuration methods to define how a path is to be secured

Method	**What it does**
access(String	Allows access if the given SpEL expression evaluates to true
anonymous()	Allows access to anonymous users
authenticated()	Allows access to authenticated users
denyAll()	Denies access unconditionally
fullyAuthenticated()	Allows access if the user is fully authenticated (not remembered)
hasAnyAuthority(String…	Allows access if the user has any of the given authorities
hasAnyRole(String…	Allows access if the user has any of the given roles
hasAuthority(String	Allows access if the user has the given authority
hasIpAddress(String	Allows access if the request comes from the given IP address

hasRole(String	Allows access if the user has the given role
not()	Negates the effect of any of the other access methods
permitAll()	Allows access unconditionally
rememberMe()	Allows access for users who are authenticated via remember-me

Spring security specific methods in SpEL

Security expression	**What it evaluates to**
authentication	The user's authentication object
denyAll	Always evaluates to false
hasAnyRole(list of roles	True if the user has any of the given roles
hasRole(role	True if the user has the given role
hasIpAddress(IP address	True if the request comes from the given IP address
isAnonymous()	True if the user is anonymous
isAuthenticated()	True if the user is authenticated
isFullyAuthenticated()	True if the user is fully authenticated (not authenticated with remember-me)
isRememberMe()	True if the user was authenticated via remember-me
permitAll	Always evaluates to true
principal	The user's principal object

SpEL expression in spring security example allow users with ROLE_USER authority to create new tacos on Tuesdays

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
       .authorizeRequests()
       .antMatchers("/design", "/orders")
       .access("hasRole('ROLE_USER') && " +
       "T(java.util.Calendar).getInstance().get("+
       "T(java.util.Calendar).DAY_OF_WEEK) == " +
       "T(java.util.Calendar).TUESDAY")
       .antMatchers("/", "/**").access("permitAll");
}
```

Spring security - get logged in user details in java methods

```
//can be used anywhere in the application
Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
```

```
User user = (User) authentication.getPrincipal();

// in controller methods we can use below annotation to get user object directly injected
@PostMapping
public String processOrder(@Valid Order order, Errors errors, SessionStatus sessionStatus,
@AuthenticationPrincipal User user) {
//..
}
```

Spring security, Spring REST security based on http request type

```
.antMatchers(HttpMethod.GET, "/api/customers").hasRole("EMPLOYEE")
.antMatchers(HttpMethod.GET, "/api/customers/**").hasRole("EMPLOYEE")
.antMatchers(HttpMethod.POST, "/api/customers").hasAnyRole("MANAGER", "ADMIN")
.antMatchers(HttpMethod.POST, "/api/customers/**").hasAnyRole("MANAGER", "ADMIN")
.antMatchers(HttpMethod.PUT, "/api/customers").hasAnyRole("MANAGER", "ADMIN")
.antMatchers(HttpMethod.PUT, "/api/customers/**").hasAnyRole("MANAGER", "ADMIN")
.antMatchers(HttpMethod.DELETE, "/api/customers/**").hasRole("ADMIN")
```

Spring method security

In a Spring web application, the application context which holds the Spring MVC beans for the dispatcher servlet is often separate from the main application context. It is often defined in a file called myapp-servlet.xml, where “myapp” is the name assigned to the Spring DispatcherServlet in web.xml.
An application can have multiple DispatcherServlets, each with its own isolated application context. The beans in these “child” contexts are not visible to the rest of the application. The “parent” application context is loaded by the ContextLoaderListener that you define in your web.xml and is visible to all the child contexts. This parent context is usually where you define your security configuration, including the <global-method-security> element).

Recommendation is applying method security at the service layer rather than on individual web controllers. (Enable method securty in RootConfig (contextLoaderListener application config) rather than on WebConfig(Dispatcher Servlet config)))

Spring method security configuration

```
@Configuration
@EnableGlobalMethodSecurity(
 prePostEnabled = true,
 securedEnabled = true,
 jsr250Enabled = true)
public class MethodSecurityConfig
 extends GlobalMethodSecurityConfiguration {
}

// The prePostEnabled property enables Spring Security pre/post annotations
// The securedEnabled property determines if the @Secured annotation should be enabled
// The jsr250Enabled property allows us to use the @RoleAllowed annotation
```

Spring method security - control access to methods based on role

```
// User having either ROLE_VIEWER or ROLE_EDITOR can invoke the isValidUsername method.
@Secured({ "ROLE_VIEWER", "ROLE_EDITOR" })
public boolean isValidUsername(String username) {
   return userRoleRepository.isValidUsername(username);
}

// RolesAllowed annotation is equivalent to @Secured annotation
// It is a java standard annotation
@RolesAllowed({ "ROLE_VIEWER", "ROLE_EDITOR" })
public boolean isValidUsername2(String username) {

}
```

Spring method security annotations taking SpEL expressions as input

Annotations	**Description**
@PreAuthorize	Restricts access to a method before invocation based on the result of evaluating an expression
@PostAuthorize	Allows a method to be invoked, but throws a security exception if the expression evaluates to false
@PostFilter	Allows a method to be invoked, but filters the results of that method based on an expression
@PreFilter	Allows a method to be invoked, but filters input prior to entering the method

Spring method security control access to methods using SpEL expressions before and after method invocation

```
// Spittr (twitter like) user can only write spittles(tweets) of 140 characters or less, but
premium users are allowed unlimited spittle lengths.
@PreAuthorize(
       "(hasRole('ROLE_SPITTER') and #spittle.text.length() <= 140)"
       +"or hasRole('ROLE_PREMIUM')")
public void addSpittle(Spittle spittle) {
// ...
}

// authorizes access if the Spittle object returned belongs to the authenticated user
// For easy access to the object returned from the secured method, Spring Security provides the
returnObject variable in SpEL
@PostAuthorize("returnObject.spitter.username == principal.username")
public Spittle getSpittleById(long id) {
       // ...
}
```

```
//@PostFilter evaluates that expression against each member of a collection being returned from the
method, removing those members for whom the expression evaluates to false.
// The filterObject referenced in the expression refers to an individual element (which you know to
be a Spittle) in the List returned from the method.
@PreAuthorize("hasAnyRole({'ROLE_SPITTER', 'ROLE_ADMIN'})")
@PostFilter( "hasRole('ROLE_ADMIN') || " + "filterObject.spitter.username == principal.name")
public List<Spittle> getOffensiveSpittles() {
	//...
}

// Spittles(tweets) can only be deleted by the user who owns them, so prefilter to exclude those
spittles not owned by the user
// @PreFilter filters those members of a collection going into the method.
// The expression will be evaluated against each item in the collection, and only those items for
whom the expression evaluates to true will remain in the list.
// The targetObject variable is another Spring Security-provided value that represents the current
list item to evaluate against
@PreAuthorize("hasAnyRole({'ROLE_SPITTER', 'ROLE_ADMIN'})")
@PreFilter( "hasRole('ROLE_ADMIN') || "
+ "targetObject.spitter.username == principal.name")
public void deleteSpittles(List<Spittle> spittles) {
	//...
}
```

Spring data

The Spring-Data is an umbrella project having many sub-projects or modules to provide uniform abstractions and uniform utility methods for the Data Access Layer in an application and support a wide range of databases and datastores. It supports both RDBMS and NoSQL databases (spring data JPA, spring-data-redis, spring-data-mongo etc)

Spring data JPA

JPA is a specification that defines an API for object-relational mappings and for managing persistent objects. Hibernate and EclipseLink are 2 popular implementations of this specification.

Spring Data JPA adds a layer on top of JPA. That means it uses all features defined by the JPA specification, especially the entity and association mappings, the entity lifecycle management, and JPA's query capabilities. On top of that, Spring Data JPA adds its own features like a no-code implementation of the repository pattern and the creation of database queries from method names. Reference: https://thorben-janssen.com/what-is-spring-data-jpa-and-why-should-you-use-it/

No code repository

The repository pattern is one of the most popular persistence-related patterns. It hides the data store specific implementation details and enables you to implement your business code on a higher abstraction level. Implementing that pattern isn't too complicated but writing the standard CRUD operations for each entity

creates a lot of repetitive code. Spring Data JPA provides you a set of repository interfaces which you only need to extend to define a specific repository for one of your entities.

Example of repository that provides methods like below

- to persist, update and remove one or multiple Author entities,
- to find one or more Authors by their primary keys,
- to count, get and remove all Authors and
- to check if an Author with a given primary key exists.

```
package org.thoughts.on.java.spring.data.repository;
import org.springframework.data.repository.CrudRepository;
import org.thoughts.on.java.spring.data.model.Author;

public interface AuthorRepository extends CrudRepository<Author, Long> {}
```

Reduced boilerplate code

Spring Data JPA provides a default implementation for each method defined by one of its repository interfaces. That means that you no longer need to implement basic read or write operations.

Generated queries

Another comfortable feature of Spring Data JPA is the generation of database queries based on method names. As long as your query isn't too complex, you just need to define a method on your repository interface with a name that starts with *find…By*

```
public interface SpitterRepository extends JpaRepository<Spitter, Long> {
   Spitter findByUsername(String username);
}
```

Nothing else needs to be done to implement findByUsername()

- Spring Data knows that this method is intended to find Spitters, because you parameterized JpaRepository with Spitter.
- The method name, findByUsername, makes it clear that this method should find Spitters by matching their username property with the username passed in as a parameter to the method.
- As return value is a single Spitter and not a collection, it knows that it should look for only one Spitter.

Points to note about JPA repository methods

- Spring Data defines a sort of miniature domain-specific language (DSL) where persistence details are expressed in repository method signatures.
- Repository methods are composed of a verb, an optional subject, the word By, and a predicate. The subject is ignored for the most part.
    - In the case of findByUsername(), the verb is find, subject if present could have been Spitter and the predicate is Username

Examples of repository methods with different operators

Query method comparison operators:

- IsAfter, After, IsGreaterThan, GreaterThan

- IsGreaterThanEqual, GreaterThanEqual
- IsBefore, Before, IsLessThan, LessThan
- IsLessThanEqual, LessThanEqual
- IsBetween, Between
- IsNull, Null
- IsNotNull, NotNull
- IsIn, In
- IsNotIn, NotIn
- IsStartingWith, StartingWith, StartsWith
- IsEndingWith, EndingWith, EndsWith
- IsContaining, Containing, Contains
- IsLike, Like
- IsNotLike, NotLike
- IsTrue, True
- IsFalse, False
- Is, Equals
- IsNot, Not

Examples:
List<Spitter> readByFirstnameOrLastname(String first, String last);
List<Spitter> readByFirstnameIgnoringCaseOrLastnameIgnoresCase(String first, String last);
List<Spitter> readByFirstnameOrLastnameAllIgnoresCase(String first, String last);
List<Spitter> readByFirstnameOrLastnameOrderByLastnameAsc(String first, String last);
List<Spitter> readByFirstnameOrLastnameOrderByLastnameAscFirstnameDesc(String first, String last);
List<Pet> findPetsByBreedIn(List<String> breed);
int countProductsByDiscontinuedTrue();
List<Order> findByShippingDateBetween(Date start, Date end);

Methods based on custom query

In situations where the desired data can't be adequately expressed in the method name, you can use the @Query annotation to provide Spring Data with the query that should be performed.

```
@Query("select s from Spitter s where s.email like '%gmail.com'")
List<Spitter> findAllGmailSpitters();

// custom method based on update query:
public class SpitterRepositoryImpl implements SpitterSweeper {
	@PersistenceContext
	private EntityManager em;
	public int eliteSweep() {
		String update =
		"UPDATE Spitter spitter " +
		"SET spitter.status = 'Elite' " +
		"WHERE spitter.status = 'Newbie' " +
		"AND spitter.id IN (" +
		"SELECT s FROM Spitter s WHERE (" +
		" SELECT COUNT(spittles) FROM s.spittles spittles) > 10000" +
		")";
```

```
            return em.createQuery(update).executeUpdate();
        }
}

public interface SpitterSweeper{
   int eliteSweep();
}

public interface SpitterRepository
extends JpaRepository<Spitter, Long>,
SpitterSweeper {
   ....
}
```

Note about implementation class prefix

Note: Spring Data JPA associates the implementation class with the interface because the implementation's name is based on the name of the interface. The "Impl" postfix is the default.(SpitterRepositorympl for SpitterRepository)

override it as follows
@EnableJpaRepositories(
basePackages="com.habuma.spittr.db",
repositoryImplementationPostfix="Helper")

<jpa:repositories base-package="com.habuma.spittr.db"
repository-impl-postfix="Helper" />

Spring data JPA @Service annotation

- @Service applied to Service implementations
- Spring will automatically register the Service implementation

Purpose of Service Layer

- Service Facade design pattern
- Intermediate layer for custom business logic
- Integrate data from multiple sources (DAO/repositories)
- Do multiple repository operations in single DB transaction using @Transactional annotation

Example

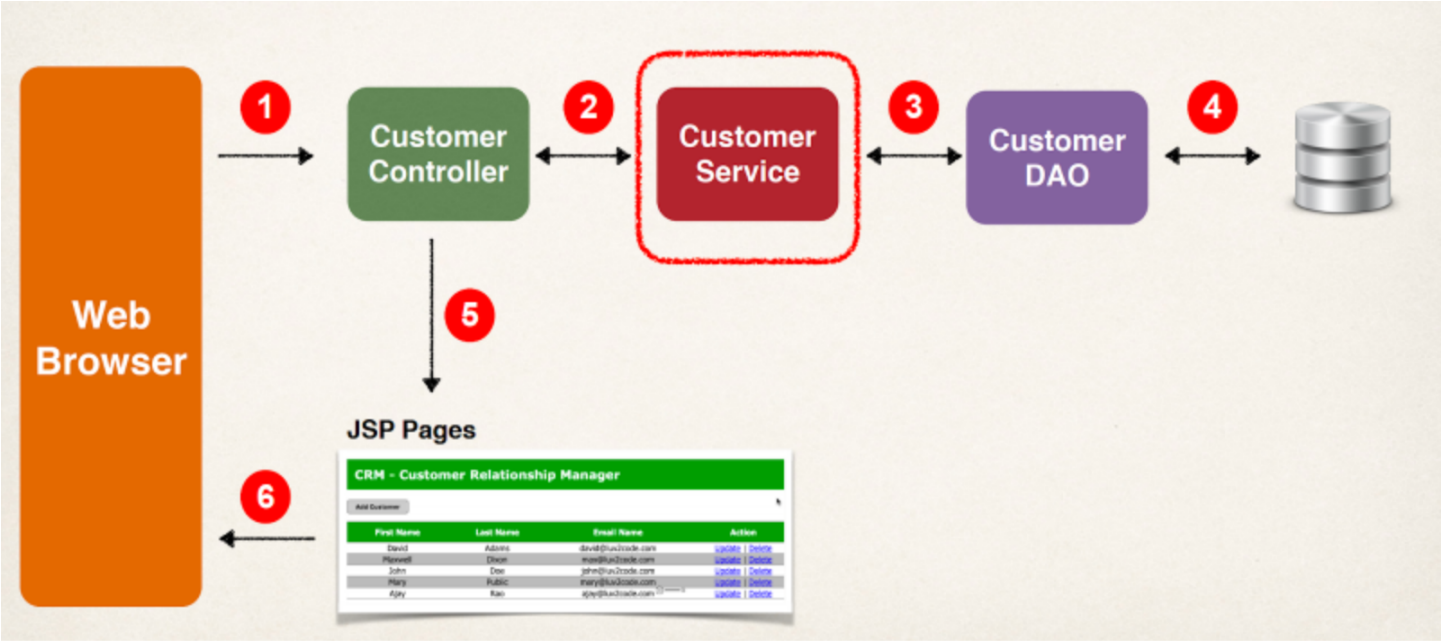
BankDAO repository has below two methods
- deposit(...)
- withdraw(...)

If we are transferring funds, we want that to run in the same transaction. By making use of @Transactional at service layer, then we can have this transactional support and both methods will run in the same transaction. This would call deposit() and withdraw(). If either of those methods failed then we'd want to roll the transaction back.

```
// Defining service is as follows

// Define service interface
```

```java
public interface CustomerService {
    public List<Customer> getCustomers();
}

// Define service implementation
@Service
public class CustomerServiceImpl implements CustomerService {

    @Autowired
    private CustomerDAO customerDAO;

    @Transactional
    public List<Customer> getCustomers() {
        //…
    }
}

// the DAO used to inject into service, does not use @Transactional
@Repository
public class CustomerDAOImpl implements CustomerDAO {

    @Autowired
    private SessionFactory sessionFactory;

    public List<Customer> getCustomers() {
        //…
    }
}
```

Spring JPA data flow



Spring REST

JSON Data Binding with Jackson

- Spring uses the Jackson Project behind the scenes
- Jackson handles data binding between JSON and Java POJO
- By default, Jackson will call appropriate getter/setter method in java POJO while binding data from json to java POJO (POJO: Plain old java object)
- When building Spring REST applications, Spring will automatically handle Jackson Integration

- JSON data being passed to REST controller is converted to POJO
- Java object being returned from REST controller is converted to JSON
- All this happens automatically behind the scenes
- Refer Spring Framework for sample code on converting json to POJO and vice versa

Maven dependency:

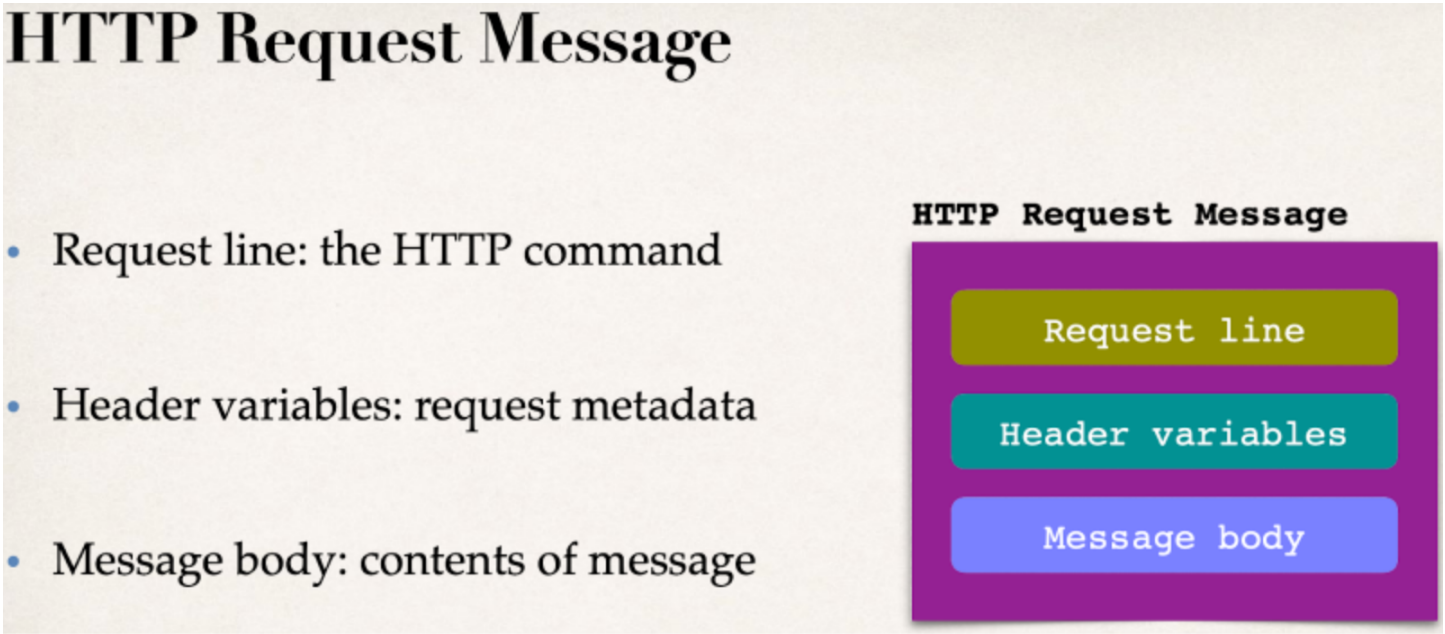
```
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
   <version>2.9.0</version>
</dependency>
```

About HTTP

HTTP methods / operations

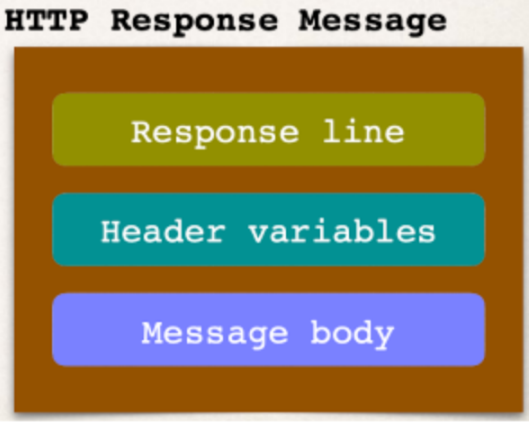
HTTP Method	CRUD Operation
POST	Create a new entity
GET	Read a list of entities or single entity
PUT	Update an existing entity
DELETE	Delete an existing entity

HTTP request/response message structure



HTTP Response Message

- Response line: server protocol and status code
- Header variables: response metadata
- Message body: contents of message



HTTP response codes

HTTP Response - Status Codes

Code Range	Description
100 - 199	Informational
200 - 299	Successful
300 - 399	Redirection
400 - 499	Client error
500 - 599	Server error

401 Authentication Required
404 File Not Found

500 Internal Server Error

HTTP MIME content type

- The message format is described by MIME content type
- Multipurpose Internet Mail-Extension
- Basic Syntax: type/sub-type
- Examples
    - text/html, text/plain

- application/json, application/xml, … etc

How Spring supports developing REST service

- Controllers can handle requests for all HTTP methods, including the four primary REST methods: GET, PUT, DELETE, PATCH and POST.
- The @PathVariable annotation enables controllers to handle requests for parameterized URLs (URLs that have variable input as part of their path).
- Resources can be represented in a variety of ways using Spring views and view resolvers, including View implementations for rendering model data as XML, JSON, Atom, and RSS.
- The representation best suited for the client can be chosen using ContentNegotiatingViewResolver.
- View-based rendering can be bypassed altogether using the @ResponseBody annotation and various HttpMethodConverter implementations.
- Similarly, the @RequestBody annotation, along with HttpMethodConverter implementations, can convert inbound HTTP data into Java objects passed in to a controller's handler methods.
- Spring applications can consume REST resources using RestTemplate.

Spring REST hello world service

```
// hello world REST service

package com.luv2code.springdemo.rest;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/test")
public class DemoRestController {

	// add code for the "/hello" endpoint

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

}
```

ResponseBody annotation

@ResponseBody annotation tells Spring that you want to send the returned object as a resource to the client, converted into some representational form that the client can accept. If the client's Accept header specifies that the client will accept application/json, and if the Jackson JSON library is in the application's classpath, then either MappingJacksonHttpMessageConverter or MappingJackson2HttpMessageConverter will be chosen depending on which version of Jackson is in the classpath

Example

```
@RequestMapping(method=RequestMethod.GET,produces="application/json")
public @ResponseBody List<Spittle>
spittles(@RequestParam(value="max",defaultValue=MAX_LONG_AS_STRING) long max,
@RequestParam(value="count", defaultValue="20") int count) {
   return spittleRepository.findSpittles(max, count);
}
```

RequestMapping and RequestBody annotation

@RequestBody tells Spring to find a message converter to convert a resource representation coming from a client into an object.

@RequestMapping has a consumes attribute set to application/json. The consumes attribute works much like the produces attribute, only with regard to the request's Content-Type header. This tells Spring that this method will only handle POST requests to /spittles if the request's Content-Type header is application/json.

Example

```
@RequestMapping(method=RequestMethod.POST, consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public @ResponseBody
Spittle saveSpittle(@RequestBody Spittle spittle) {
   return spittleRepository.save(spittle);
}
```

[Difference between the annotations @GetMapping and @RequestMapping(method = RequestMethod.GET)]

- @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).
- @GetMapping is the newer annotaion.
- @GetMapping we can apply only on method level and @RequestMapping annotation we can apply on both class as well as method level

ResponseEntity annotation

ResponseEntity is a wrapper for HTTP response object.
Controller methods can return a ResponseEntity, which is an object that carries metadata (such as headers and the status code) about a response in addition to the object(response body) to be converted to a resource representation.

Example 1

An exception handler can deal with the error cases, leaving the handler methods to focus on the happy path.

```
//example code demonstrating usage of ResponseEntity
//Error is a POJO with attributes code(int) and message(string)
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<?> spittleById(@PathVariable long id) {
   Spittle spittle = spittleRepository.findOne(id);
   if (spittle == null) {
       Error error = new Error(4, "Spittle [" + id + "] not found");
       return new ResponseEntity<Error>(error, HttpStatus.NOT_FOUND);
   }
   return new ResponseEntity<Spittle>(spittle, HttpStatus.OK);
}
```

Example 2

Using ResponseEntity to set any response header:

```
@RequestMapping(method=RequestMethod.POST,consumes="application/json")
public ResponseEntity<Spittle> saveSpittle( @RequestBody Spittle spittle, UriComponentsBuilder ucb)
{
```

```
   Spittle spittle = spittleRepository.save(spittle);
   HttpHeaders headers = new HttpHeaders();
   URI locationUri = ucb.path("/spittles/")            // calculate the location URI
   .path(String.valueOf(spittle.getId()))
   .build()
   .toUri();
   headers.setLocation(locationUri);
   ResponseEntity<Spittle> responseEntity = new ResponseEntity<Spittle>(spittle, headers,
HttpStatus.CREATED);
   return responseEntity;
}
```

ResponseStatus annotation

If we know that a controller method is always supposed to return same http response status, then we can use ResponseStatus annotation.

For example, error handler method always returns an Error and always responds with an HTTP status code of 404 (Not Found)

```
@ExceptionHandler(SpittleNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public @ResponseBody Error spittleNotFound(SpittleNotFoundException e) {
   long spittleId = e.getSpittleId();
   return new Error(4, "Spittle [" + spittleId + "] not found");
}
```

Spring REST service returning POJO response by converting it to JSON

```
// REST service returning POJO
// RESTController service returning POJO response
// global exception handlers return http 404 if student is not found or if path parameter is
invalid
package com.luv2code.springdemo.rest;

import java.util.ArrayList;
import java.util.List;
import javax.annotation.PostConstruct;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.luv2code.springdemo.entity.Student;

@RestController
@RequestMapping("/api")
public class StudentRestController {

   private List <Student> theStudents;

   // define @PostConstruct to load the student data ... only once!
   @PostConstruct
   public void loadData() {
       theStudents = new ArrayList < > ();
       theStudents.add(new Student("Poornima", "Patel"));
       theStudents.add(new Student("Mario", "Rossi"));
       theStudents.add(new Student("Mary", "Smith"));
   }
```

```
    // define endpoint for "/students" - return list of students
    @GetMapping("/students")
    public List < Student > getStudents() {
        return theStudents;
    }

    // define endpoint for "/students/{studentId}" - return student at index
    @GetMapping("/students/{studentId}")
    public Student getStudent(@PathVariable int studentId) {
        // just index into the list ... keep it simple for now
        // check the studentId against list size
        if ((studentId >= theStudents.size()) || (studentId < 0)) {
            throw new StudentNotFoundException("Student id not found - " + studentId);
        }
        return theStudents.get(studentId);
    }

    //Can have @ExceptionHandler annotated methods here as well to have exception handling per class
basis
}

// Controller advice for global exception handling
package com.luv2code.springdemo.rest;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class StudentRestExceptionHandler {

    // add exception handling code here
    // Add an exception handler using @ExceptionHandler
    @ExceptionHandler
    public ResponseEntity < StudentErrorResponse > handleException(StudentNotFoundException exc) {

        // create a StudentErrorResponse
        StudentErrorResponse error = new StudentErrorResponse();
        error.setStatus(HttpStatus.NOT_FOUND.value());
        error.setMessage(exc.getMessage());
        error.setTimeStamp(System.currentTimeMillis());

        // return ResponseEntity
        return new ResponseEntity < > (error, HttpStatus.NOT_FOUND);
    }

    // add another exception handler ... to catch any exception (catch all)
    @ExceptionHandler
    public ResponseEntity < StudentErrorResponse > handleException(Exception exc) {

        // create a StudentErrorResponse
        StudentErrorResponse error = new StudentErrorResponse();
        error.setStatus(HttpStatus.BAD_REQUEST.value());
        error.setMessage(exc.getMessage());
        error.setTimeStamp(System.currentTimeMillis());

        // return ResponseEntity
        return new ResponseEntity < > (error, HttpStatus.BAD_REQUEST);
    }
}

// Student is POJO with firstName and lastName string attribute
package com.luv2code.springdemo.entity;

public class Student {

    private String firstName;
    private String lastName;
```

```
    public Student() { }

    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName = firstName; }
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
}

//  student not found exception which just has message
package com.luv2code.springdemo.rest;

public class StudentNotFoundException extends RuntimeException {

        public StudentNotFoundException(String message, Throwable cause) {
                super(message, cause);
        }

        public StudentNotFoundException(String message) {
                super(message);
        }

        public StudentNotFoundException(Throwable cause) {
                super(cause);
        }
}

// error response POJO

package com.luv2code.springdemo.rest;
public class StudentErrorResponse {

    private int status;
    private String message;
    private long timeStamp;

    public StudentErrorResponse() { }

    public StudentErrorResponse(int status, String message, long timeStamp) {
        this.status = status;
        this.message = message;
        this.timeStamp = timeStamp;
    }

    public int getStatus() { return status; }
    public void setStatus(int status) { this.status = status; }
    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }
    public long getTimeStamp() { return timeStamp; }
    public void setTimeStamp(long timeStamp) { this.timeStamp = timeStamp; }
}
```

Spring REST - Annotations for supporting different HTTP methods

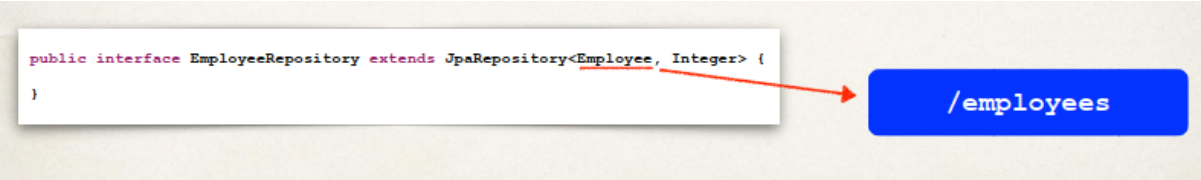
```
@RestController
@RequestMapping("/api/v1")
public class EmployeeController {
    @Autowired
    private EmployeeRepository employeeRepository;

    @GetMapping("/employees")
    public List < Employee > getAllEmployees() {
        return employeeRepository.findAll();
```

```
    }

    @GetMapping("/employees/{id}")
    public ResponseEntity < Employee > getEmployeeById(@PathVariable(value = "id") Long employeeId)
    throws ResourceNotFoundException {
        Employee employee = employeeRepository.findById(employeeId)
            .orElseThrow(() - > new ResourceNotFoundException("Employee not found for this id :: " +
employeeId));
        return ResponseEntity.ok().body(employee);
    }

    @PostMapping(path = "/employees",
        consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_JSON_VALUE)
    public Employee createEmployee(@Valid @RequestBody Employee employee) {
        return employeeRepository.save(employee);
    }

    @PutMapping("/employees/{id}")
    public ResponseEntity < Employee > updateEmployee(@PathVariable(value = "id") Long employeeId,
        @Valid @RequestBody Employee employeeDetails) throws ResourceNotFoundException {
        Employee employee = employeeRepository.findById(employeeId)
            .orElseThrow(() - > new ResourceNotFoundException("Employee not found for this id :: " +
employeeId));

        employee.setEmailId(employeeDetails.getEmailId());
        employee.setLastName(employeeDetails.getLastName());
        employee.setFirstName(employeeDetails.getFirstName());
        final Employee updatedEmployee = employeeRepository.save(employee);
        return ResponseEntity.ok(updatedEmployee);
    }

    @DeleteMapping("/employees/{id}")
    public Map < String, Boolean > deleteEmployee(@PathVariable(value = "id") Long employeeId)
    throws ResourceNotFoundException {
        Employee employee = employeeRepository.findById(employeeId)
            .orElseThrow(() - > new ResourceNotFoundException("Employee not found for this id :: " +
employeeId));

        employeeRepository.delete(employee);
        Map < String, Boolean > response = new HashMap < > ();
        response.put("deleted", Boolean.TRUE);
        return response;
    }

    @PatchMapping("/employees/{id}/{firstName}")
    public ResponseEntity < Employee > updateEmployeePartially(@PathVariable Long id, @PathVariable
String firstName) {
        try {
            Employee employee = employeeRepository.findById(id).get();
            employee.setFirstName(firstName);
            return new ResponseEntity < Employee > (employeeRepository.save(employee),
HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity < > (HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}

// REST end point which can return both json and xml response
// -----------------------
@Controller
@RequestMapping(path = "/", produces = MediaType.APPLICATION_JSON_VALUE)
public class HomeController
{
    @PostMapping(path = "/members")
    public void addMember_V1(@RequestBody Member member) {
        //code
    }
    @PostMapping(path = "/members", produces = MediaType.APPLICATION_XML_VALUE)
```

```
    public void addMember_V2(@RequestBody Member member) {
        //code
    }
}
```

Spring data REST

- Spring data REST automatically creates REST endpoints for your repositories
- Spring Data REST will scan your project for JpaRepository
- Expose REST APIs for each entity type for your JpaRepository
- By default, Spring Data REST will create endpoints based on entity type
    - Simple pluralized form
    - First character of Entity type is lowercase
    - Then just adds an "s" to the entity



    - We can specify custom endpoint entity name using below annotation

```
@RepositoryRestResource(path="members")
public interface EmployeeRepository extends JpaRepository<Employee, Integer> { }
```

- Just need to add dependency in maven / gradle, no coding required

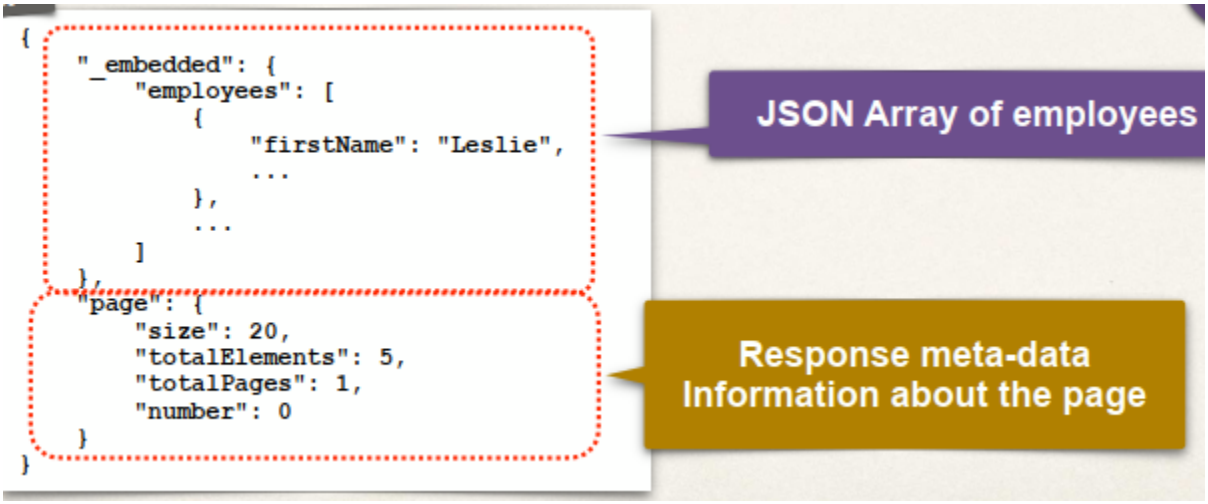
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

    - 

HATEOAS

- Spring Data REST endpoints are HATEOAS compliant
    - HATEOAS: Hypermedia as the Engine of Application State
- Hypermedia-driven sites provide information to access REST interfaces
    - Think of it as meta-data for REST data
- Example REST response from: GET /employees/3, (get single employee)

- Example REST response from: GET /employees (get list of employees)



REST configuration, pagination and sorting

Pagination

- By default, Spring Data REST will return the first 20 elements
  - Page size = 20
- You can navigate to the different pages of data using query param
- Pages are zero based
  - http://localhost:8080/employees?page=0
  - http://localhost:8080/employees?page=1

Configuration

Following application properties are available for configuration

Name	Description
spring.data.rest.base-path	Base path used to expose repository resources
spring.data.rest.default-page-size	Default size of pages
spring.data.rest.max-page-size	Maximum size of pages

Sorting

- You can sort by the property names of your entity
- In our Employee example, we have: firstName, lastName and email
- Sort by last name (ascending is default)
    - http://localhost:8080/employees?sort=lastName
- Sort by first name, descending
    - http://localhost:8080/employees?sort=firstName,desc
- Sort by last name, then first name, ascending
    - http://localhost:8080/employees?sort=lastName,firstName,asc

Spring Boot

Java Spring Boot (Spring Boot) is a tool that makes developing web application and microservices with Spring Framework faster and easier through three core capabilities:

1. Autoconfiguration
2. An opinionated approach to configuration
3. The ability to create **standalone** applications along with embedded server

Spring boot and spring framework comparison

	Spring	**Spring Boot**
What is it?	An open-source web application framework based on Java.	An extension or module built on the Spring framework.
What does it do?	Provides a flexible, completely configurable environment using tools and libraries of prebuilt code to create customized, loosely coupled web apps.	Provides the ability to create, standalone Spring applications that can just run immediately without the need for annotations, XML configuration, or writing lots of additional code.
When should I use it?	Use Spring when you want: ● Flexibility ● An **unopinionated** approach ● To remove dependencies from your custom code. ● To implement a very unique configuration. ● To develop enterprise applications.	Use Spring Boot when you want: ● Ease of use ● An **opinionated** approach. ● To get quality apps running quickly and reduce development time. ● To avoid writing boilerplate code or configuring XML. ● To develop REST APIs.
What's its key feature?	Dependency injection	Autoconfiguration
Does it have embedded servers?	No. In Spring, you'll need to set up the servers explicitly.	Yes, Spring Boot comes with built-in HTTP servers like Tomcat and Jetty.

How is it configured?	The Spring framework provides flexibility, but its configuration has to be built manually.	Spring Boot configures Spring and other third-party frameworks automatically by the default “**convention over configuration”** principle.
Do I need to know how to work with XML?	In Spring, knowledge of XML configuration is required.	Spring Boot does not require XML configuration.
Are there CLI tools for dev/testing apps?	The Spring framework alone doesn't provide CLI tools for developing or testing apps.	As a Spring module, Spring Boot has a CLI tool for developing and testing Spring-based apps.
Does it work from an opinionated or unopinionated approach?	**Unopinionated**	**Opinionated**

Reference: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-java-spring-boot/

Opinionated approach

An opinionated approach takes the position that there is one way that is significantly easier than all the others. By design, **the software limits designers, encouraging them into doing things in that prescribed way**.

It **provides a well-paved path, a best practice that will work for most people in most situations**. The app is written sticking closely to these best practices and widespread conventions. An opinionated approach makes collaboration and getting help with a coding project much easier—other developers who have experience with that framework will have immediate familiarity with the new app and can jump right in.

Unopinionated approach

An unopinionated approach is adopted if all solutions require roughly the same amount of effort or complexity. It takes the position that there's no one right way to arrive at a solution to a problem. Rather, it provides flexible tools that can be used to solve the problem in many ways. Unopinionated frameworks have the **benefit of providing lots of flexibility in development and they put more of the control in developers' hands**. The main disadvantage with so much flexibility is that the **developer has more decisions to make and may end up having to write more code** because the framework is so open-ended and well, unopinionated.

What is convention over configuration?

Convention over configuration, sometimes called coding by convention, is a concept used in application frameworks to **reduce the number of decisions that a developer has to make**. It adheres to the "don't repeat yourself" principle to avoid writing redundant code. Coding by convention strives to maintain flexibility while allowing a developer to only write code for the unconventional aspects of the app they're creating. When the desired behavior of the app matches the conventions established, the app will just run by default without having to write configuration files. The developer will only need to explicitly write configuration files if the desired behavior strays from the "convention."

Advantages of using spring boot

- Make it easier to get started with Spring development
- Minimize the amount of manual configuration
- Perform autoconfiguration based on property files and JAR classpath
- Help to resolve dependency conflicts (Maven or Gradle)
- Uses spring framework behind the scenes
- Provide an embedded HTTP server so you can get started quickly
    - Tomcat, Jetty, Undertow etc.
    - No need to install a server separately

Spring Initializr

- Helps to quickly create a starter Spring Boot project
- Select your dependencies
- Can create a Maven or Gradle project
- Import the project into your IDE
    - Eclipse, IntelliJ, NetBeans etc
- https://start.spring.io/

Sample maven pom file for minimal spring boot project

```
<?xml version="1.0" encoding="UTF-8"?>
<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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>3.0.4</version>
       <relativePath/> <!-- lookup parent from repository -->
   </parent>
   <groupId>com.luv2code.springboot.demo</groupId>
   <artifactId>mycoolapp</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <name>mycoolapp</name>
   <description>Demo project for Spring Boot</description>
   <properties>
       <java.version>17</java.version>
   </properties>
   <dependencies>
       <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-web</artifactId>
       </dependency>
       <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-test</artifactId>
           <scope>test</scope>
       </dependency>
   </dependencies>
   <build>
       <plugins>
           <plugin>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-maven-plugin</artifactId>
           </plugin>
       </plugins>
   </build>
</project>
```

Advantages of Maven

- Maven will go out and download the JAR files of direct dependencies and supporting / secondary dependencies of the project for you automatically and Maven will make those JAR files available during compile/run
- Maven has standard directory structure
    - This helps in easily finding code, properties files, unit tests, web files etc
- Most major IDEs have built-in support for Maven
    - Eclipse, IntelliJ, NetBeans etc
- Maven projects are portable
    - IDEs can easily read/import Maven projects
    - Developers can easily share projects between IDEs
- Once you learn Maven, you can join a new project and be productive
- You can build and run a project with minimal local configuration

Spring boot directory structure

```
src
   pom.xml              (maven build xml or maybe use gradle)
   main
       java             (Java source code)
       resources        (Properties / config files used by the application)
           static       (by default, spring boot will load static resources from this directory)
           templates    (spring boot will load templates from this directory. eg: thymeleaf,
freemarker and mustache)
   test
       java             (Unit testing source code)
```

Spring boot starters

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

- Spring-boot-starter-web imports dependencies like below
    - spring-web
    - spring-webmvc
    - hibernate-validator
    - tomcat
    - Json etc
- Save's the developer from having to list all of the individual dependencies
- Also, makes sure you have compatible versions
- Also has spring-boot-maven-plugin, which provides maven goal to run spring boot application
    - Using maven wrapper mvnw
    - ./mvnw package
    - ./mvnw spring-boot:run

Things provided by spring boot starter parent (spring-boot-starter-parent)

- Maven defaults defined in the Starter Parent like
  - Default compiler level
  - UTF-8 source encoding etc
- No need to list versions for dependencies
- Use version on parent only
  - spring-boot-starter-* dependencies inherit version from parent
  - Default configuration of Spring Boot plugin
- Default configuration of Spring Boot maven plugin (which provides spring-boot:run goal)

Full dependency tree

https://gist.github.com/manhhavu/0c680ee1fd955f160c8c2f5983aaa90d

```
+--- org.springframework.boot:spring-boot-starter-web: -> 1.4.3.RELEASE
|    +--- org.springframework.boot:spring-boot-starter:1.4.3.RELEASE
|    |    +--- org.springframework.boot:spring-boot:1.4.3.RELEASE
|    |    |    +--- org.springframework:spring-core:4.3.5.RELEASE
|    |    |    \--- org.springframework:spring-context:4.3.5.RELEASE
|    |    |         +--- org.springframework:spring-aop:4.3.5.RELEASE
|    |    |         |    +--- org.springframework:spring-beans:4.3.5.RELEASE
|    |    |         |    |    \--- org.springframework:spring-core:4.3.5.RELEASE
|    |    |         |    \--- org.springframework:spring-core:4.3.5.RELEASE
|    |    |         +--- org.springframework:spring-beans:4.3.5.RELEASE (*)
|    |    |         +--- org.springframework:spring-core:4.3.5.RELEASE
|    |    |         \--- org.springframework:spring-expression:4.3.5.RELEASE
|    |    |              \--- org.springframework:spring-core:4.3.5.RELEASE
|    |    +--- org.springframework.boot:spring-boot-autoconfigure:1.4.3.RELEASE
|    |    |    \--- org.springframework.boot:spring-boot:1.4.3.RELEASE (*)
|    |    +--- org.springframework.boot:spring-boot-starter-logging:1.4.3.RELEASE
|    |    |    +--- ch.qos.logback:logback-classic:1.1.8
|    |    |    |    +--- ch.qos.logback:logback-core:1.1.8
|    |    |    |    \--- org.slf4j:slf4j-api:1.7.21 -> 1.7.22
|    |    |    +--- org.slf4j:jcl-over-slf4j:1.7.22
|    |    |    |    \--- org.slf4j:slf4j-api:1.7.22
|    |    |    +--- org.slf4j:jul-to-slf4j:1.7.22
|    |    |    |    \--- org.slf4j:slf4j-api:1.7.22
|    |    |    \--- org.slf4j:log4j-over-slf4j:1.7.22
|    |    |         \--- org.slf4j:slf4j-api:1.7.22
|    |    +--- org.springframework:spring-core:4.3.5.RELEASE
|    |    \--- org.yaml:snakeyaml:1.17
|    +--- org.springframework.boot:spring-boot-starter-tomcat:1.4.3.RELEASE
|    |    +--- org.apache.tomcat.embed:tomcat-embed-core:8.5.6
|    |    +--- org.apache.tomcat.embed:tomcat-embed-el:8.5.6
|    |    \--- org.apache.tomcat.embed:tomcat-embed-websocket:8.5.6
|    |         \--- org.apache.tomcat.embed:tomcat-embed-core:8.5.6
|    +--- org.hibernate:hibernate-validator:5.2.4.Final
|    |    +--- javax.validation:validation-api:1.1.0.Final
|    |    +--- org.jboss.logging:jboss-logging:3.2.1.Final -> 3.3.0.Final
|    |    \--- com.fasterxml:classmate:1.1.0 -> 1.3.3
|    +--- com.fasterxml.jackson.core:jackson-databind:2.8.5
|    |    +--- com.fasterxml.jackson.core:jackson-annotations:2.8.0 -> 2.8.5
|    |    \--- com.fasterxml.jackson.core:jackson-core:2.8.5
|    +--- org.springframework:spring-web:4.3.5.RELEASE
|    |    +--- org.springframework:spring-aop:4.3.5.RELEASE (*)
|    |    +--- org.springframework:spring-beans:4.3.5.RELEASE (*)
|    |    +--- org.springframework:spring-context:4.3.5.RELEASE (*)
|    |    \--- org.springframework:spring-core:4.3.5.RELEASE
|    \--- org.springframework:spring-webmvc:4.3.5.RELEASE
|         +--- org.springframework:spring-aop:4.3.5.RELEASE (*)
|         +--- org.springframework:spring-beans:4.3.5.RELEASE (*)
|         +--- org.springframework:spring-context:4.3.5.RELEASE (*)
|         +--- org.springframework:spring-core:4.3.5.RELEASE
|         +--- org.springframework:spring-expression:4.3.5.RELEASE (*)
|         \--- org.springframework:spring-web:4.3.5.RELEASE (*)
```

Popular spring boot starters

spring-boot-starter	Core starter, including auto-configuration support, logging and YAML
spring-boot-starter-web	Building web apps, includes validation, REST.Uses Tomcat as default embedded server
spring-boot-starter-security	Adding Spring Security support
spring-boot-starter-data-jpa	Spring database support with JPA and Hibernate
spring-boot-starter-data-rest	Starter for exposing Spring Data repositories over REST using Spring Data REST
spring-boot-starter-data-jdbc	Starter for using Spring Data JDBC
spring-boot-starter-thymeleaf	Starter for building MVC web applications using Thymeleaf views
spring-boot-starter-freemarker	Starter for building MVC web applications using FreeMarker views
spring-boot-starter-actuator	Starter for using Spring Boot's Actuator which provides production ready features to help you monitor and manage your application
spring-boot-starter-webflux	Starter for building WebFlux applications using Spring Framework's Reactive Web support
spring-boot-starter-batch	Starter for using Spring Batch
spring-boot-starter-data-mongodb-reactive	Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive
spring-boot-starter-data-neo4j	Starter for using Neo4j graph database and Spring Data Neo4j
spring-boot-starter-web-services	Starter for using Spring Web SOAP Services

Spring boot environment and application properties

The Spring environment abstraction is a one-stop shop for any configurable property. It abstracts the origins of properties so that beans needing those properties can consume them from Spring itself. The Spring environment pulls from several property sources, including

- JVM system properties
- Operating system environment variables
- Command-line arguments
- Application property configuration files

Properties are considered in the following order

1. Devtools global settings properties on your home directory (~/.spring-boot-devtools.properties when devtools is active).
2. @TestPropertySource annotations on your tests.
3. properties attribute on your tests. Available on @SpringBootTest and the test annotations for testing a particular slice of your application.

4. Command line arguments.
5. Properties from SPRING_APPLICATION_JSON (inline JSON embedded in an environment variable or system property).
6. ServletConfig init parameters.
7. ServletContext init parameters.
8. JNDI attributes from java:comp/env.
9. Java System properties (System.getProperties()).
10. OS environment variables.
11. A RandomValuePropertySource that has properties only in random.*.
12. Profile-specific application properties outside of your packaged jar (application-{profile}.properties and YAML variants).
13. Profile-specific application properties packaged inside your jar (application-{profile}.properties and YAML variants).
14. Application properties outside of your packaged jar (application.properties and YAML variants).
15. Application properties packaged inside your jar (application.properties and YAML variants).
16. @PropertySource annotations on your @Configuration classes. Please note that such property sources are not added to the Environment until the application context is being refreshed. This is too late to configure certain properties such as logging.* and spring.main.* which are read before refresh begins.
17. Default properties (specified by setting SpringApplication.setDefaultProperties)

Order of loading of properties file

SpringApplication loads properties from application.properties files in the following locations and adds them to the Spring Environment. The list is ordered by precedence (properties defined in locations higher in the list override those defined in lower locations).

- A /config subdirectory of the current directory
- The current directory
- A classpath /config package
- The classpath root

Spring boot relaxed rules for binding Environment properties

acme.my-project.person.first-name	Kebab case, which is recommended for use in .properties and .yml files.
acme.myProject.person.firstName	Standard camel case syntax.
acme.my_project.person.first_name	Underscore notation, which is an alternative format for use in .properties and .yml files.
ACME_MYPROJECT_PERSON_FIRSTNAME	Upper case format, which is recommended when using system environment variables.

Spring boot overriding properties from command line

```
java -jar springbootapp_v1.jar --acme.my-project.person.first-name=nameForCommandline
-acme.my-project.person.last-name=lnameFrmCmdline
```

Spring boot John thompson's pragmatic guide for using properties from various places

- Favor using application.properties or application.yml in packaged JAR (or WAR)
- Use profile specific properties or YAML files for profile specific properties
- For deployments, override properties that change with environment variables
- Typically 70-80% of values do not change, only override what is needed
- Environment variables offer a secure way of setting sensitive values such as password

spring boot specifying property in different formats

```
#in .properties file
server.port=9090

#in yaml/yml file
server:
   port: 9090

#as operating system variable
#the naming style is slightly different to accommodate restrictions placed on environment variable
names by the
#operating system. Spring is capable of handling it and interpret's SERVER_PORT as server.port
export SERVER_PORT=9090

#set a property named greeting.welcome to echo the value of another property named
spring.application.name
#To achieve this, you could use the ${} placeholder markers
greeting:
   welcome: ${spring.application.name}
```

Spring boot important application properties

```
#setting log level for package
#package name is specified under logging.level
logging.level.com.luv2code=INFO

#setting log file name
logging.file=MyApp.log
#other logging properties
#set root logging level to WARN but log spring security logs at DEBUG level
logging:
   path: /var/logs/
   file: TacoCloud.log
   level:
       root: WARN
       org.springframework.security: DEBUG

#HTTP server port
server.port=7070

#Context path of the application
server.servlet.context-path=/my-first-app

#Default HTTP session timeout
#Default timeout is 30m
server.servlet.session.timeout=15m

#Setting the property value to true means that all the beans in the application will use lazy
initialization.
spring.main.lazy-initialization=true
```

```
#Actuator endpoints
#Endpoints to include by name or wildcard
management.endpoints.web.exposure.include=*

#Endpoints to exclude by name or wildcard
management.endpoints.web.exposure.exclude=beans,mapping

#Base path for actuator endpoints
management.endpoints.web.base-path=/actuator

#actuator info
info.app.name=My Super Cool App
info.app.description=A crazy and fun app, yoohoo!
info.app.version=1.0.0

#Spring security
#Default username
spring.security.user.name=admin

#Password for default user
spring.security.user.password=topsecret

#Spring data
#JDBC URL of the database
spring.datasource.url=jdbc:mysql://localhost:3306/ecommerce

#Login username and password of the database
spring.datasource.username=scott
spring.datasource.password=tiger

#in yaml
spring:
 datasource:
   url: jdbc:mysql://localhost/tacocloud
   username: tacodb
   password: tacopassword
   driver-class-name: com.mysql.jdbc.Driver

#specifying datasource from JNDI lookup
spring:
 datasource:
jndi-name: java:/comp/env/jdbc/tacoCloudDS

#embedded server ssl configuration
server:
 port: 8443
 ssl:
   key-store: file:///path/to/mykeys.jks
   key-store-password: letmein
   key-password: letmein


#specifying the database initialization scripts to run when the application starts

spring:
 datasource:
schema:
- order-schema.sql
- ingredient-schema.sql - taco-schema.sql
- user-schema.sql
```

```
data:
- ingredients.sql

# enable h2 in-memory database console
# available at /h2-console
spring.h2.console.enabled=true
```

Spring boot creating property holding object using constructor binding

https://springframework.guru/immutable-property-binding/

```
// Note: fields have no setter methods and are immutable
// here property values for username, password etc are injected in constructor
// we can also use @PropertySource annotation to consider property values from given properties
file
@ConstructorBinding
@ConfigurationProperties("guru")
public class SfgConstructorConfig {
   private final String username;
   private final String password;
   private final String jdbcurl;

   public SfgConstructorConfig(String username, String password, String jdbcurl) {
       this.username = username; this.password = password; this.jdbcurl = jdbcurl;
   }

   public String getUsername() { return username; }
   public String getPassword() { return password; }
   public String getJdbcurl() { return jdbcurl; }
}

@EnableConfigurationProperties(SfgConstructorConfig.class)
@ImportResource("classpath:sfgdi-config.xml")
@Configuration
public class GreetingServiceConfig {

   @Bean
   FakeDataSource fakeDataSource(SfgConstructorConfig sfgConstructorConfig){
       FakeDataSource fakeDataSource = new FakeDataSource();
       fakeDataSource.setUsername(sfgConstructorConfig.getUsername());
       fakeDataSource.setPassword(sfgConstructorConfig.getPassword());
       fakeDataSource.setJdbcurl(sfgConstructorConfig.getJdbcurl());
       return fakeDataSource;
   }
   // other methods
}
```

Profiles

Profiles are a type of conditional configuration where different beans, configuration classes, and configuration properties are applied or ignored based on what profiles are active at runtime.One way to define profile-specific properties is to create yet another YAML or properties file containing only the properties for profile.

The name of the file should follow this convention: application-{profile name}.yml OR application-{profile name}.properties

Another way to specify profile-specific properties works only with YAML configuration. It involves placing profile-specific properties alongside non-profiled properties in application.yml, separated by three hyphens and the spring.profiles property to name the profile.

```
logging:
 level:
   tacos: DEBUG

---
```

```
spring:
 profiles: prod
 datasource:
   url: jdbc:mysql://localhost/tacocloud
   username: tacouser
   password: tacopassword
logging:
 level:
   tacos: WARN
```

Activating profiles

```
#in application.yml

spring:
 profiles:
   active:
   - prod
   - audit
   - ha

#via environment variable
export SPRING_PROFILES_ACTIVE=prod,audit,ha

#via command-line argument
java -jar taco-cloud.jar --spring.profiles.active=prod
```

Conditionally creating beans based on active profiles

```
@Bean
@Profile({"dev", "qa"})
//activate bean when prod profile is not active
//@Profile("!prod")
//activate bean when neither prod profile nor qa profile is active
//@Profile({"!prod", "!qa"})
public CommandLineRunner dataLoader(IngredientRepository repo,
   //methods ...
}
```

Spring boot about @SpringBootApplication annotation

@SpringBootApplication is composed of the following annotations:

@EnableAutoConfiguration	Enables Spring Boot's auto-configuration support
@ComponentScan	Enables component scanning of current package. Also recursively scans sub-packages
@Configuration	Able to register extra beans with @Bean or import other configuration classes

```
//Explicitly specifying third party packages for component scanning in @SpringBootApplication
annotation
@SpringBootApplication(
   scanBasePackages={"com.luv2code.springboot.demo.mycoolapp",
   "org.acme.iot.utils",
   "edu.cmu.wean"})
```

Spring boot dev tools

Spring boot dev tools are additional set of tools that can make the application development experience more pleasant
It provides below features without needing to write additional code (Reference: baeldung spring boot dev tools)

- Automatically restarts your application when code is updated
    - Need to have “build project automatically” setting turned on in IDE’s like intellij, eclipse etc
- Has sensible property defaults for iteration of code development and testing
    - Example: By default caching of page templates is on which is used by template engines like thymeleaf. But during development, it's more important to see the changes as quickly as possible. Caching can be disabled for *thymeleaf* using the property *spring.thymeleaf.cache=false* in the *application.properties* file. We do not need to do this manually, introducing this *spring-boot-devtools* does this automatically for us.
- Automatic restart - Whenever files change in the classpath, applications using *spring-boot-devtools* will cause the application to restart. The benefit of this feature is the time required to verify the changes made is considerably reduced
- Live Reload - *spring-boot-devtools* module includes an embedded LiveReload server that is used to trigger a browser refresh when a resource is changed.For this to happen in the browser we need to install the LiveReload plugin one such implementation is Remote Live Reload for Chrome.
- Remote Debugging via HTTP (Remote Debug Tunnel) - *spring-boot-devtools* provides out of the box remote debugging capabilities via HTTP, to have this feature it is required that *spring-boot-devtools* are packaged as part of the application.

Maven dependency

```
   <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-devtools</artifactId>
       <optional>true</optional>
   </dependency>
```

Spring boot actuator

Spring boot actuator is an additional module in spring boot which provides production ready features like

- Monitoring our app
- Gathering metrics
- Understanding traffic
- Know state of the database
- expose operational information about the running application
    - health, metrics, info, dump, env, etc
- Works with both spring mvc and spring webflux (reactive)

Maven dependency

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

Important changes after version 2.x

Reference: baeldung spring boot actuator

- Unlike in previous versions, **Actuator comes with most endpoints disabled**

    - the only two available by default are */health* and */info*
- by default, all Actuator endpoints are now placed under the */actuator* path.
    - we can tweak this path using the new property *management.endpoints.web.base-path*
- Actuator now shares the security config with the regular App security rules, so the security model is dramatically simplified.
    - Security config to allow free access to all actuator endpoints

```
@Bean
public SecurityWebFilterChain securityWebFilterChain(
 ServerHttpSecurity http) {
   return http.authorizeExchange()
     .pathMatchers("/actuator/**").permitAll()
     .anyExchange().authenticated()
     .and().build();
}
```

List of actuator endpoints

Note: For almost all of them we have to invoke HTTP GET

ID	Description
beans	Displays a complete list of all the Spring beans in your application.
env	Exposes properties from Spring's ConfigurableEnvironment.
/env/{name}	Retrieves a specific environment value by name.
info	Displays arbitrary application info.
loggers	Shows and modifies the configuration of loggers in the application.
health	Shows application health information.
httpexchanges	Displays HTTP exchange information (by default, the last 100 HTTP request-response exchanges). Requires an HttpExchangeRepository bean.
metrics	Shows "metrics" information for the current application.
/metrics/{name}	Reports an individual application metric by name.
mappings	Displays a collated list of all @RequestMapping paths.
sessions	Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Requires a servlet-based web application that uses Spring Session.
auditevents	Exposes audit events information for the current application. Requires an AuditEventRepository bean.
caches	Exposes available caches.
conditions	Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match.

configprops	Displays a collated list of all @ConfigurationProperties.
flyway	Shows any Flyway database migrations that have been applied. Requires one or more Flyway beans.
liquibase	Shows any Liquibase database migrations that have been applied. Requires one or more Liquibase beans.
(POST) /shutdown	Lets the application be gracefully shutdown. Only works when using jar packaging. /shutdown is disabled by default.
startup	Shows the startup steps data collected by the ApplicationStartup. Requires the SpringApplication to be configured with a BufferingApplicationStartup.
threaddump	Performs a thread dump.
integrationgraph	Shows the Spring Integration graph. Requires a dependency on spring-integration-core.
quartz	Shows information about Quartz Scheduler jobs.
scheduledtasks	Displays the scheduled tasks in your application.

Further customization

- For security purposes, we might choose to expose the actuator endpoints over a non-standard port, the *management.port* property can easily be used to configure that.
- We can change the *management.address* property to restrict where the endpoints can be accessed from over the network

```
#port used to expose actuator
management.port=8081

#CIDR allowed to hit actuator
management.address=127.0.0.1

#Whether security should be enabled or disabled altogether
management.security.enabled=false
#If the application is using Spring Security, we can secure these endpoints by defining the
default security properties (username, password, and role) in the application.properties file
security.user.name=admin
security.user.password=secret
management.security.role=SUPERUSER
```

Spring boot security

- Spring Security defines a framework for security
- Implemented using Servlet filters in the background
    - Servlet Filters are used to pre-process / post-process web requests
    - Servlet Filters can route web requests based on security logic
    - Spring provides a bulk of security functionality with servlet filters
- Two methods of securing an app: declarative and programmatic
    - Programmatic
        - Spring Security provides an API for custom application coding
        - Provides greater customization for specific app requirements
    - Declarative

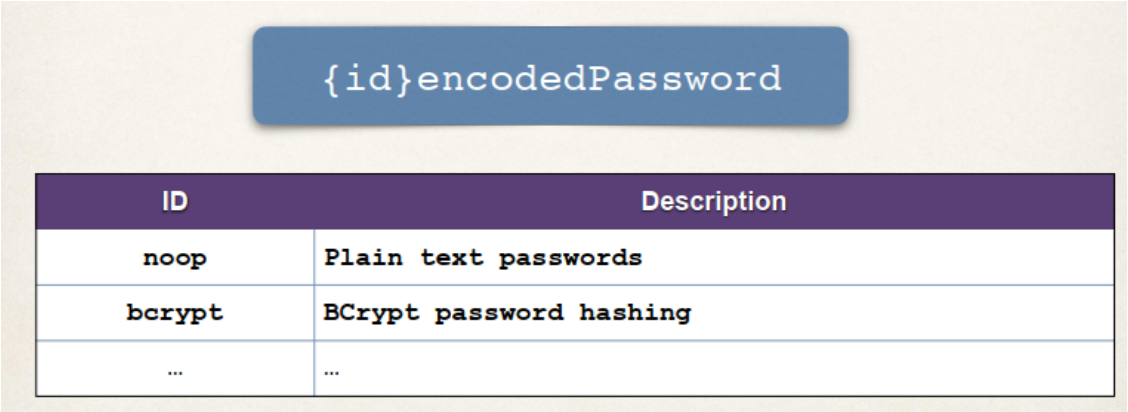
- Define application's security constraints in configuration
  - All Java config: @Configuration
- Provides separation of concerns between application code and security

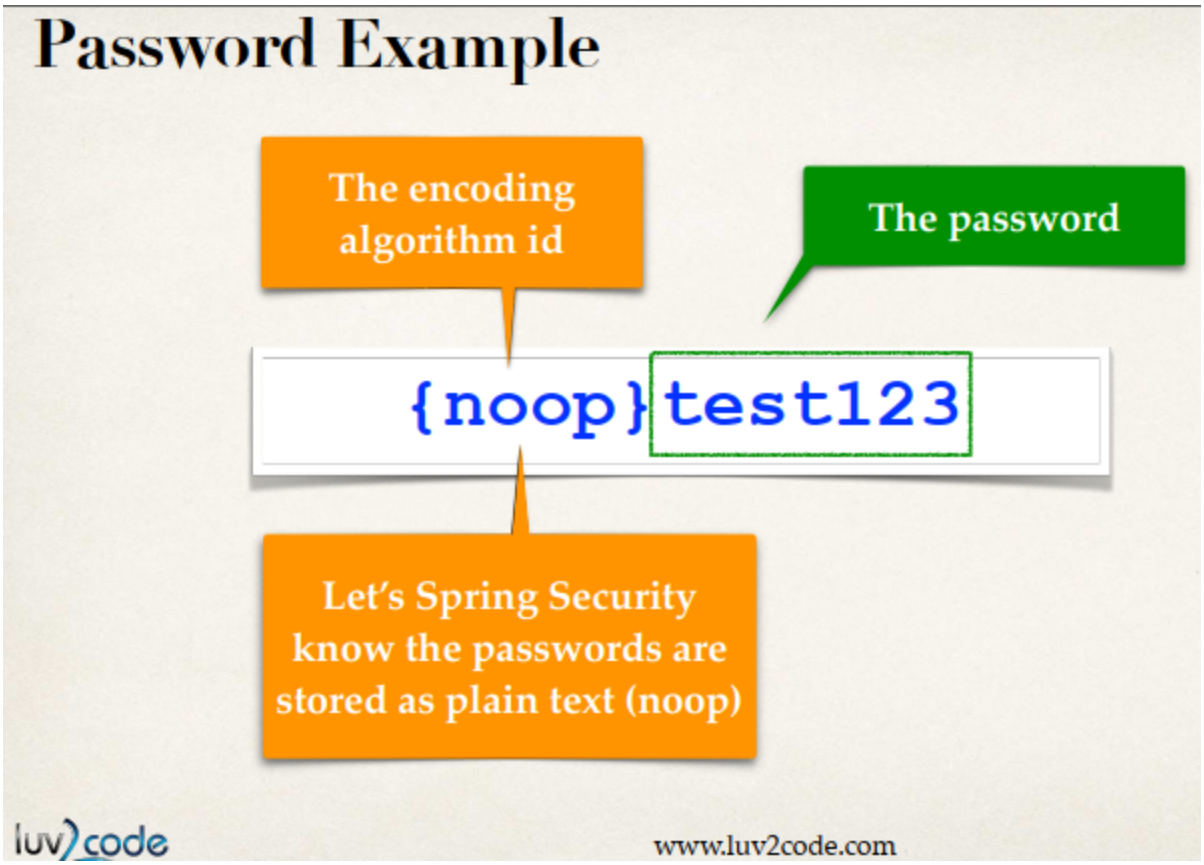
Security auto configuration

- Adding spring-boot-starter-security dependency will automatically secure all endpoints for application
- By default, the Authentication gets enabled for the Application. Also, content negotiation is used to determine if basic or formLogin should be used.
- Now when you access your application. Spring Security will prompt for login
- If a default password is not specified, then a password is generated and is printed in console log
- You can override default user name and generated password
  spring.security.user.name=scott
  spring.security.user.password=test123

Password storage

In Spring Security, passwords are stored using a specific format.





About password handling

- It's better to always use https for spring boot application in production because otherwise password is transmitted in plain text format. Reference: stackoverflow question
- Its better to use default passwordEncoder bean, which is DelegatingPasswordEncoder
    - Reference: another stackoverflow question
- Currently bcrypt scheme is the default password encoding scheme when default passwordEncoder bean is used, but it can change. Since the id of encoding scheme is stored along with password, there is no problem while decoding existing stored passwords in the database. Reference: Password Storage History

Specifying security configuration in spring boot - sample code

Reference: chad darby luv2code spring boot course github repo

```
package com.luv2code.springboot.cruddemo.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.provisioning.JdbcUserDetailsManager;
import org.springframework.security.provisioning.UserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

import javax.sql.DataSource;

@Configuration
public class DemoSecurityConfig {

    // add support for JDBC ... no more hardcoded users :-)
```

```
    @Bean
    public UserDetailsManager userDetailsManager(DataSource dataSource) {

        JdbcUserDetailsManager jdbcUserDetailsManager = new JdbcUserDetailsManager(dataSource);

        // define query to retrieve a user by username
        jdbcUserDetailsManager.setUsersByUsernameQuery(
                "select user_id, pw, active from members where user_id=?");

        // define query to retrieve the authorities/roles by username
        jdbcUserDetailsManager.setAuthoritiesByUsernameQuery(
                "select user_id, role from roles where user_id=?");

        return jdbcUserDetailsManager;
    }


    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        http.authorizeHttpRequests(configurer ->
                configurer
                        .requestMatchers(HttpMethod.GET, "/api/employees").hasRole("EMPLOYEE")
                        .requestMatchers(HttpMethod.GET, "/api/employees/**").hasRole("EMPLOYEE")
                        .requestMatchers(HttpMethod.POST, "/api/employees").hasRole("MANAGER")
                        .requestMatchers(HttpMethod.PUT, "/api/employees").hasRole("MANAGER")
                        .requestMatchers(HttpMethod.DELETE, "/api/employees/**").hasRole("ADMIN")
        );

        // use HTTP Basic authentication
        http.httpBasic();

        // disable Cross Site Request Forgery (CSRF)
        // in general, not required for stateless REST APIs that use POST, PUT, DELETE and/or PATCH
        http.csrf().disable();

        return http.build();
    }

/*
    @Bean
    public InMemoryUserDetailsManager userDetailsManager() {

        UserDetails john = User.builder()
                .username("john")
                .password("{noop}test123")
                .roles("EMPLOYEE")
                .build();

        UserDetails mary = User.builder()
                .username("mary")
                .password("{noop}test123")
                .roles("EMPLOYEE", "MANAGER")
                .build();

        UserDetails susan = User.builder()
                .username("susan")
                .password("{noop}test123")
                .roles("EMPLOYEE", "MANAGER", "ADMIN")
                .build();

        return new InMemoryUserDetailsManager(john, mary, susan);
    }
*/
}
```