Spring 4

Text: EN

Software: EN

REALDOLMEN
to get there, together

Table of contents

1. Spring 4 Workshop

1.1. Goal

Learn how to use the Spring Framework to simplify Java Enterprise development

1.2. Agenda

- Introduction
- Wiring Beans
- Aspect Oriented Programming
- Spring Persistence (JDBC / JPA)
- Spring Data
- Transaction Support
- Spring Web MVC
- Spring Test
- Spring Boot
- Spring Boot with Groovy

2. Introduction

2.1. What is Spring?

- Spring's fundamental mission: Simplify Java Development
- Spring is an open source framework
  - Described in by Rod Johnson's book "Expert One-on-One: J2EE Design and Development"
- Spring was created to address the complexity of enterprise application development
  - Use plain-vanilla JavaBeans to achieve things previously only possible with EJB
  - Any Java application can benefit from Spring: simplicity, testability and loose coupling

2.2. History of Spring

- Spring had a tremendous impact on enterprise application development
  - It has become a de facto standard for many Java projects
  - It had an impact on the evolution on the standard specifications and frameworks
- Spring continues to evolve and improve upon itself
  - Making difficult development tasks simpler
  - Empower Java developers with new and innovative features
  - Paving new trails in Java application development
  - Spring progresses into areas where JEE is just starting or isn't innovating at all

2.3. Exciting new features in Spring 4

- Emphasis on Java-based configuration
- Conditional configuration and profiles
- Enhancements and improvements to Spring MVC, especially for REST support
- Using Thymeleaf with Spring web applications as alternative to JSP
- Enabling Spring Security with Java-based configuration
- Using Spring Data to automatically generate repository implementations at runtime (for JPA, MongoDB and Neo4j)
- New declarative caching support
- Asynchronous web messaging with WebSocket and STOMP
- Spring Boot, a new approach to working with Spring

2.4. Spring strategies

- Everything Spring does boils down to:
  - Lightweight and minimally invasive development with POJOs
  - Loose coupling through dependency injection and interface orientation
  - Declarative programming through aspects and common conventions
  - Eliminating boilerplate code with aspects and templates
- As introduction, let us see some examples of the above strategies

2.5. The power of POJOs

- Spring avoids (as much as possible) littering your application code with its API
  - Spring almost never forces you to implement Spring-specific interfaces or extend a Spring-specific class
  - Classes in a Spring-based application often have no indication that they're being used by Spring

- At most, a class may be annotated with one of Spring's annotations, but it's a POJO

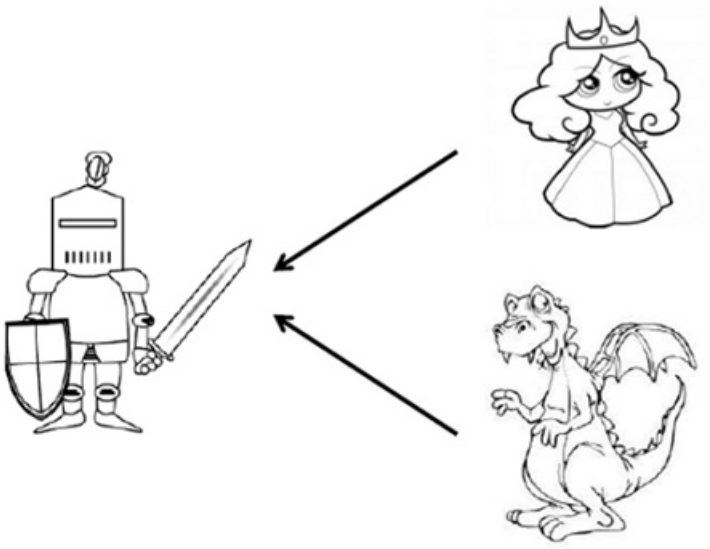
```
public class HelloWorldBean { // this is all you need for Spring
  public String sayHello() {
    return "Hello World";
  }
}
```

2.6. Dependency injection

- Makes your code simpler, easier to understand and easier to test
- Consider following example

```
public class DamselRescuingKnight implements Knight {
  private RescueDamselQuest quest;

  public DamselRescuingKnight() {
    this.quest = new RescueDamselQuest(); // this is a dependency
  }

  public void embarkOnQuest() {
    quest.embark();
  }
}
```

- The previous class is tightly coupled
  - If you need any other quest... the knight will have to sit it out
- This code is also very difficult to test, to reuse and to understand
  - If anything goes wrong when embarking on a quest... which class is responsible?
  - How can you test the `DamselRescuingKnight` in isolation?
- We need coupling or else nothing would happen in our code
- Coupling needs to be carefully managed!
- Definition of dependency injection
  - Objects are given their dependencies at creation time by some third party that coordinates each object in the system
  - Objects are not expected to create or obtain their dependencies themselves



- In the example below, the knight receives a quest at construction time
  - This is called constructor injection

- Since the given quest implements the `Quest` interface, the knight can embark on any quest: `RescueDamselQuest`, `SlayDragonQuest`, ...

```
public class BraveKnight implements Knight {
  private Quest quest;

  public BraveKnight(Quest quest) { // Quest is injected
    this.quest = quest;
  }
  public void embarkOnQuest() {
    quest.embark();
  }
}
```

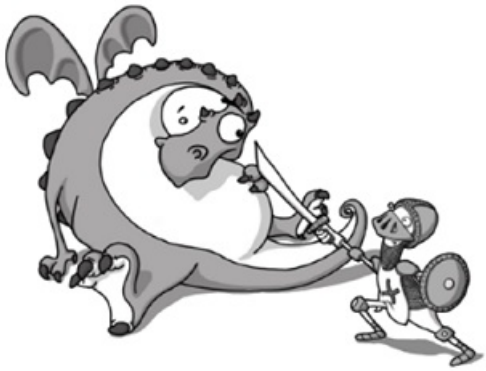
- The key benefit is loose coupling
  - If an object only knows about its dependencies by their interface, then the dependency can be swapped out with a different implementation
  - The depending object would not even know the difference
- Loose coupling can then make code easier to test
  - One common reason to swap out a dependency, is to introduce a mock implementation
  - This allows you to test the code in isolation, in a true unit test!
- Example using JUnit and Mockito

```
public class BraveKnightTest {
  @Test
  public void knightShouldEmbarkOnQuest() {
    Quest mockQuest = mock(Quest.class); // create mock
    BraveKnight knight = new BraveKnight(mockQuest); // inject mock
    knight.embarkOnQuest();
    verify(mockQuest, times(1)).embark();
  }
}
```

2.7. Wiring

- Suppose you want the knight to embark on a quest to slay a dragon

```
public class SlayDragonQuest implements Quest {
  private PrintStream stream;
  public SlayDragonQuest(PrintStream stream) {
    this.stream = stream;
  }
  public void embark() {
    stream.println("Going to slay the dragon!");
  }
}
```



- How do you give this quest to the knight? How do you give the `PrintStream` to the `SlayDragonQuest`?
- Wiring is the act of creating associations between application components
- In Spring, there are many ways of to wire components together

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="knight" class="com.acme.knights.BraveKnight">
    <constructor-arg ref="quest" /> <!--injecting the quest bean -->
```

```
  </bean>
  <bean id="quest" class="com.acme.knights.SlayDragonQuest">
    <constructor-arg value="#{T(System).out}" /> <!--using Spring EL -->
  </bean>
</beans>
```

- An equivalent in Java configuration

```
@Configuration
public class KnightConfig {
  @Bean
  public Knight knight() {
    return new BraveKnight(quest()); // injecting the quest bean
  }

  @Bean
  public Quest quest() {
    return new SlayDragonQuest(System.out);
  }
}
```

- Whether you use XML or Java-based configuration, the benefits are the same

2.8. Seeing it work

- To start Spring with configuration, we need to start an application context
  - This is a framework object responsible to load bean definitions and wire them

```
public class KnightMain {
  public static void main(String[] args) throws Exception {
    // load the Spring context
    ClassPathXmlApplicationContext context =
      new ClassPathXmlApplicationContext("/META-INF/spring/knight.xml");
    Knight knight = context.getBean(Knight.class); // get the knight bean
    knight.embarkOnQuest(); // use knight
    context.close();
  }
}
```

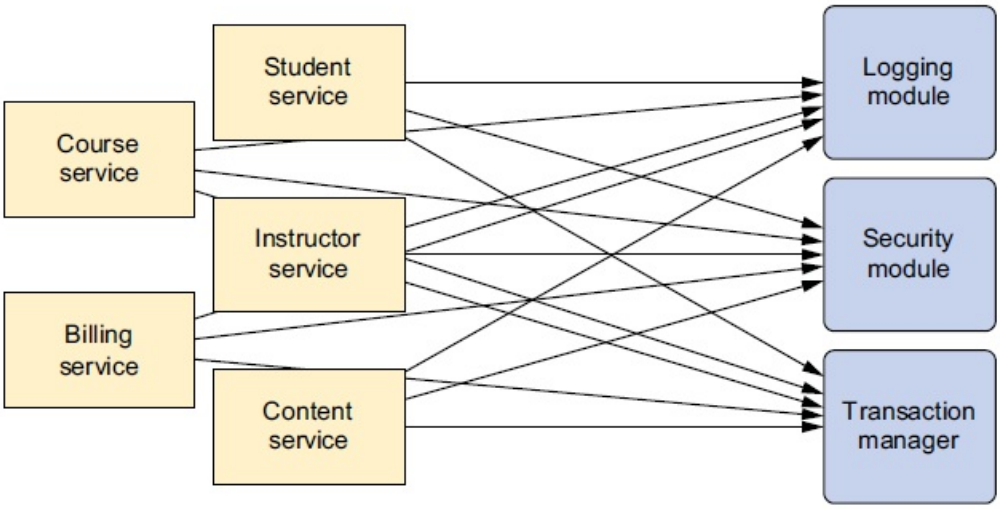
- For use with a Java configuration and Spring Boot, use another application context implementation

```
@SpringBootApplication
public class KnightMain{
  public static void main(String[] args) throws Exception {
    // load the Spring context
    ConfigurableApplicationContext context = SpringApplication.run(KnightMain.class);
    Knight knight = context.getBean(Knight.class); // get the knight bean
    knight.embarkOnQuest(); // use knight
    context.close();
  }
}
```

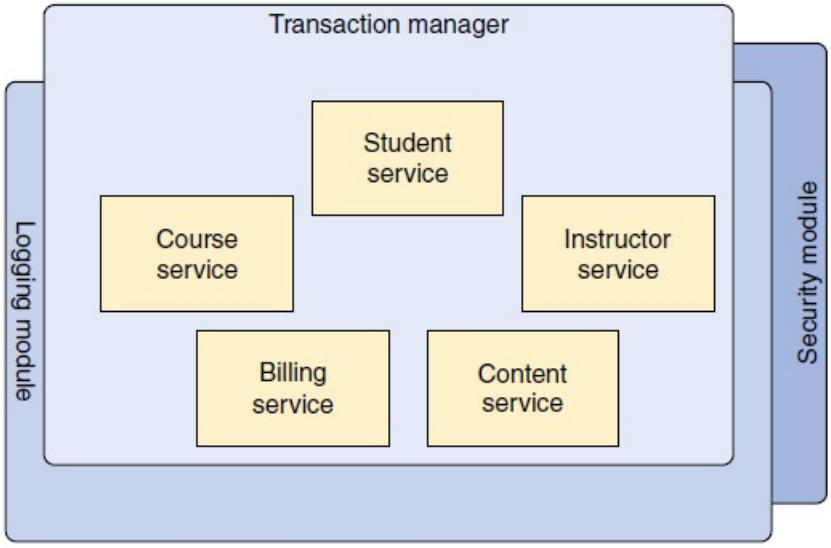
2.9. Applying aspects

- Aspect Oriented Programming (AOP) separates cross-cutting concerns
  - It enables you to capture functionality that is used throughout your application in reusable components
- System services are usually cross-cutting
  - Logging, transaction management, security
  - They often find their way into components whose core responsibility is elsewhere
  - These services cut across multiple components in the system

- Cross-cutting introduces complexity
  - Code to implement these services is duplicated across multiple components
    - Even if you abstract the concern in a separate module, the method call is still duplicated
  - Your components are littered with code that is not aligned with their core functionality
- Illustration
  - In the below example, the business objects on the left are too intimately involved with the system services on the right



- AOP makes it possible to modularize these services and then apply them declaratively to the components they should affect
  - The core application does not even know these services exist



- Adding a minstrel to the knight

```
public class Minstrel {
  private PrintStream stream;

  public Minstrel(PrintStream stream) {
    this.stream = stream;
  }

  public void singBeforeQuest() {
```

```
    stream.println("Fa la la, the knight is so brave!");
  }

  public void singAfterQuest() {
    stream.println("Tee heehee, the brave knight did embark on a quest!");
  }
}
```

- Should you inject the minstrel into the knight?
- It is not the knight's concern to manage the minstrel and his state
  - The minstrel should do his job without having to be asked to
  - The knight should not have to remind the minstrel
- Using AOP you can separate the minstrel code
  - The minstrel can then sing about the knight's quest, while the knight is freed of the minstrel management
- We will keep minstrel as a POJO, but declare it as aspect
  - We configure when the minstrel should sing...

```
@Aspect // an aspect
public class Minstrel {
  private PrintStream stream;

  public Minstrel(PrintStream stream) {
    this.stream = stream;
  }

  @Pointcut("execution(* *.embarkOnQuest(..))") // a pointcut
  private void embarksOnQuest(){
  }

  @Before("embarksOnQuest()") // a before advice
  public void singBeforeQuest() {
    stream.println("Fa la la, the knight is so brave!");
  }

  @After("embarksOnQuest()") // an after advice
  public void singAfterQuest() {
    stream.println("Tee hee hee, the brave knight did embark on a quest!");
  }
}
```

- We just need a few extra lines of configuration to enable AOP ... and done!

```
@Configuration
@EnableAspectJAutoProxy // enabling AOP
public class KnightConfig {

  // assume knight and quest beans still present...

  @Bean
  public Minstrel minstrel() { // adding the minstrel
    return new Minstrel(System.out);
  }
}
```

- Of course you can also use an XML file to configure AOP instead

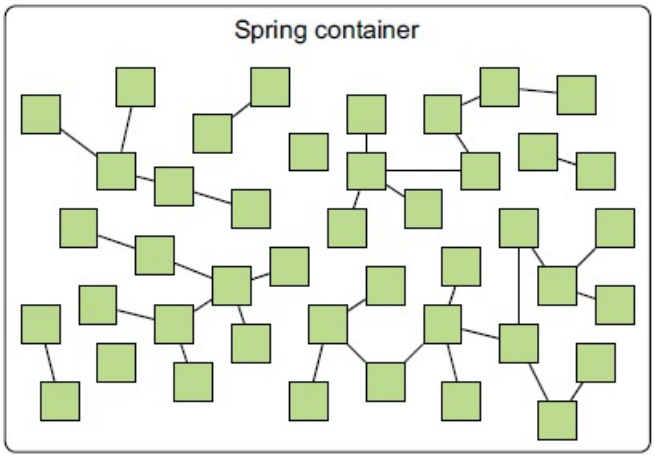
2.10. Templating

- There are a lot of places where the Java API involves boilerplate code
  - This is code that you often have to write over and over again to accomplish common and otherwise simple tasks
  - Examples: JDBC, JNDI, consuming REST Services, ...

- Spring seeks to eliminate boilerplate code by encapsulating it in templates
  - As developer you can focus on business instead of technology to make the code work
  - Templates eliminate much of the ceremony required when using APIs

2.11. Spring containers

- Application objects live in a Spring container
  - The container creates the objects, wires them together, configures them and manages their lifecycle
- Spring comes with several container implementations



- Bean factories
  - Are the simplest containers, providing basic support for dependency injection
  - `org.springframework.beans.factory.BeanFactory` interface
- Application contexts
  - Provide additional application-framework services
  - `org.springframework.context.ApplicationContext` interface
- Application contexts are preferred, as bean factories are too low-level for most applications
- Application context implementations
  - `AnnotationConfigApplicationContext`
    - Loads Spring from one or more Java-based configuration classes
  - `AnnotationConfigWebApplicationContext`
    - As above, but for web applications
  - `ClassPathXmlApplicationContext`
    - Loads a context definition from one or more XML files located in the classpath
  - `FileSystemXmlApplicationContext`
    - As above, but the files are located on the file system
  - `XmlWebApplicationContext`
    - As above, but contained in a web application
  - `ConfigurableApplicationContext`
    - A context returned by Spring Boot
- Examples

```
ApplicationContext context = new
```

```
    FileSystemXmlApplicationContext("c:/knight.xml");
```

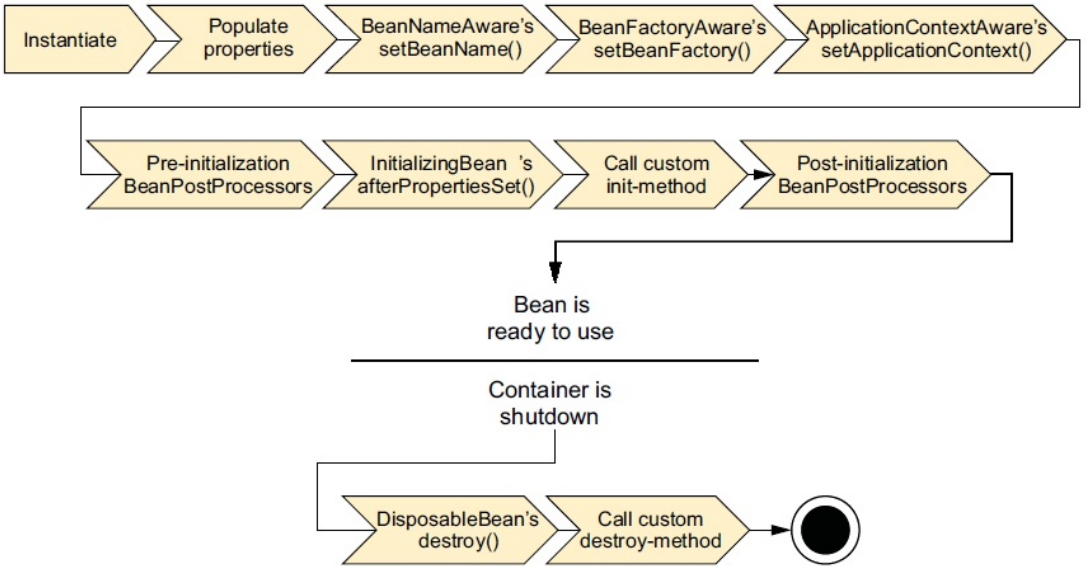
```
ApplicationContext context = new
    ClassPathXmlApplicationContext("knight.xml");
```

```
ApplicationContext context = new
    AnnotationConfigApplicationContext(KnightConfig.class);
```

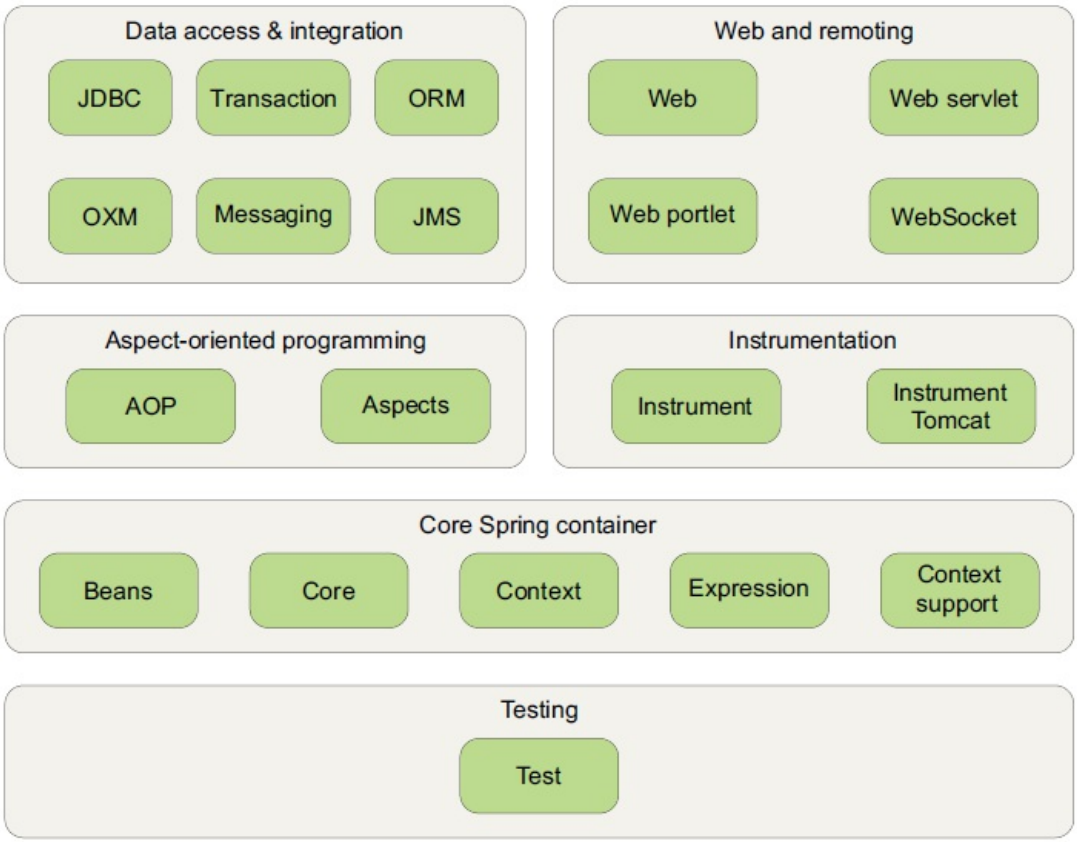
```
ConfigurableApplicationContext context =
    SpringApplication.run(KnightConfig.class, args);
```

2.12. Bean lifecycle

- A bean goes through several steps between creation and destruction in the Spring container



2.13. Spring framework modules



- Core Spring container
  - The centerpiece of the Spring framework
  - Contains the bean factory and application context implementations
- Spring AOP module
  - Provides rich support for AOP in Spring
- Data access and integration
  - Support for JDBC, ORM, JMS, OXM and transactions
- Web and remoting
  - Includes MVC support, remoting (RMI, Hessian, Burlap, JAX-WS, HTTPInvoker) and REST
- Instrumentation
  - Provides support for adding agents to the JVM (weaving agent for Tomcat bytecode instrumentation)
- Testing
  - Module dedicated to testing Spring applications

2.14. Spring portfolio

- Next to the default modules, Spring includes several frameworks and libraries build on top of core Spring
- Spring Web Flow
  - Supports building conversational, flow-based web applications, build on Spring MVC
- Spring Web Services
  - Offers a contract-first web services model
- Spring Security
  - Declarative security mechanism for Spring applications, implemented with AOP

- Spring Integration
  - Implementation of several common integration patterns
- Spring Batch
  - For bulk operations on data
- Spring Data
  - Makes it easy to work with all kinds of databases in Spring
  - Adds support for NoSQL databases
- Spring Social
  - Helps to connect Spring applications with REST APIs, social or not
- Spring Mobile
  - An extension to Spring MVC to support development of mobile web applications
- Spring for Android
  - Brings the simplicity of Spring to the development of native Android applications
- Spring Boot
  - Makes developing with Spring easier using automatic configuration techniques

2.15. What's new in Spring 4?

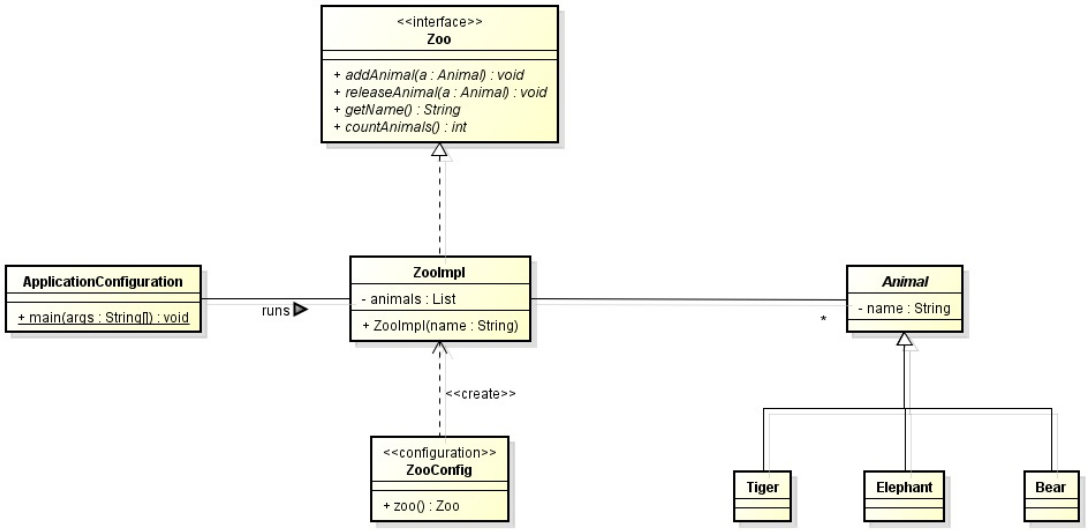
- Support for JSR-356: Java API for WebSocket
- Support for a higher level message-oriented programming model on top of WebSockets(using SockJSand STOMP protocol)
- Support for Java 8 features, including lambda's
- Support for JSR-310: Date and Time API
- Smoother programming model for applications developed in Groovy
- Generalized support for conditional bean creation
- Asynchronous implementation of the Spring RestTemplate
- Support for many JEE specs: JMS 2.0, JTA 1.2, JPA 2.1 and Bean Validation 1.1

2.16. Exercise: Introduction

- Open the "introduction" exercise provided by the Teacher
  - Explore the project to familiarize yourself with the code
- Create an interface for a Zoo in the package service
  - `void addAnimal(Animal animal)`
  - `void releaseAnimal(Animal animal)`
  - `boolean accept(Visitor visitor)`
  - `String getName()`
  - `int countAnimals()`
- Add `Animal` classes in a `domain` package
  - Animals have a `name` property (add getters and setters)
  - `Animal` is `abstract`
  - Add some concrete animal classes to the `domain` package
- Create a new `ZooImpl` implementation class that implements the `Zoo` interface
  - Add a collection of animals and implement the interface methods
- Create a class `ZooConfig` in a `config` package

- Configure the `ZooImpl` as a bean, give it a name with its constructor and add some "exotic" animals

- Start Spring using the `main` method
  - Fetch the configured `Zoo`
  - Print out its name and number of animals



3. Wiring Beans

3.1. Wiring beans

- Any application is made up of several objects that must work together to meet some business goal
  - These objects must be aware of one another and communicate with one another to get their jobs done
- This requires coupling
  - Objects need to create associations via constructor or lookup
  - This leads to complicated code and makes objects highly coupled
  - Highly coupled objects are hard to reuse and hard to test
- In Spring, the container gives an object the references it needs
  - The creation of associations between objects is called dependency injection and is commonly referred to as wiring
  - Dependency injection is the most elemental thing Spring does

3.2. Spring configuration options

- While Spring does dependency injection, developers need to tell Spring which beans to create and how to wire them
  - This requires configuration
- Spring configuration has several options
  - Explicit configuration in XML
  - Explicit configuration in Java
  - Implicit bean discovery and automatic wiring
- There is some overlap in what each configuration technique offers
  - Choice is simply a matter of personal taste, mix-and-match is possible
  - Recommendation: use automatic configuration as much as possible, then JavaConfig, fall back to XML when there is no JavaConfig option available
- During this course, we will focus on automatic and JavaConfig

3.3. Automatically wiring beans

- Spring's automatic configuration focuses on ease of use
  - Component scanning: automatically discover beans in the application context
  - Autowiring: automatically satisfy bean dependencies
- This helps keep configuration to a minimum
- Example: `Walkman` and `Cassette`
  - Using an interface, you keep the coupling to a minimum
  - It still requires an implementation


```
public interface Cassette {
  void play();
}
```

- Implementing the interface

```
@Component
public class AwesomeMixVolume1 implements Cassette {
  private String title = "Awesome Mix Volume 1";
  private String artist = "Various Artists";
  public void play() {
    System.out.println("Playing " + title + " by " + artist);
```

```
  }
}
```

- The `@Component` identifies the class as a Spring bean
- Enabling auto-discovery of beans requires some configuration

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

- `@ComponentScan` annotation enables component scanning
  - By default, `@ComponentScan` scans the same package as the configuration class, including sub-packages
  - It will find the `Cassette` implementation and create a bean for it in Spring
- To test the configuration, we can write a simple JUnit test

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=WalkmanConfig.class)
public class WalkmanTest{

  @Autowired
  private Cassette cassette;

  @Test
  public void cassetteShouldNotBeNull() {
    assertNotNull(cassette);
  }
}
```

- Explaining the test
  - The `SpringJUnit4ClassRunner` creates a Spring `ApplicationContext` as the test starts
  - `@ContextConfiguration` informs the loader which configuration to load
  - Because of the `@ComponentScan`, the `Cassette` bean will be created
  - To prove it, the bean gets `@Autowire`-d into the test
  - With a simple test method, we assert the cassette property is not null
- Any classes with `@Component` will be created as beans
  - This scales well from a few beans to many!

3.4. Component scanning

- Beans get an `id`
  - The default `id` is derived from the name of the class, starting with a lowercase
- If you want to give the bean another `id`, pass it to the `@Component`

```
@Component("guardiansOfTheGalaxySoundtrack")
public class AwesomeMixVolume1 implements Cassette {
  // ...
}
```

- You can also use the `@Named` annotation from CDI
  - In most common cases, these annotations are interchangeable

```
@Named("guardiansOfTheGalaxySoundtrack")
public class AwesomeMixVolume1 implements Cassette {
  // ...
}
```

- Spring has several specialization annotations to indicate components

- They are called stereotypes
  - `@Component`: indicates a bean for component scanning
  - `@Controller`: indicates a bean that serves as controller in Spring MVC
  - `@Repository`: indicates a bean is used for storage, retrieval and search
  - `@Service`: indicates a standalone, stateless bean that offers operations
- These annotations are all similar, but differentiating can be useful
  - They demarcate application layers
  - You can write aspects based on the presence of these annotations
  - `@Repository` beans get `Exception` translation (more on that later)
- You can set the base package in the `@ComponentScan`
  - This enables you to keep all the configuration in a package of its own
  - `basepackages` is plural, so you can add multiple strings surrounded with braces

```
@Configuration
@ComponentScan("musicplayer") // or @ComponentScan(basePackages={"musicplayer"})
public class WalkmanConfig{}
```

- Setting package names as strings is not very type-safe
  - You can also specify the packages to scan by using classes in the packages
  - Whatever packages those classes are in, will be used as base package
  - It is recommended to consider empty marker interfaces in the packages to be scanned

```
@Configuration
@ComponentScan(basePackageClasses={Walkman.class, MovieBox.class})
public class WalkmanConfig{}
```

3.5. Autowiring with annotations

- Let Spring automatically satisfy a bean's dependencies by finding other beans in the application context that match
  - Use the `@Autowired` annotation

```
@Component
public class Walkman implements MediaPlayer {
  private Cassette cassette;

  @Autowired
  public Walkman(Cassette cassette) {
    this.cassette = cassette;
  }

  public void play() {
    cassette.play();
  }
}
```

- `@Autowired` can be put on constructors and setters, or any other method
  - Spring will attempt to satisfy the dependency expressed in the method's parameters
  - Assuming that one and only one bean matches, that bean will be wired in

```
@Autowired
public void setCassette(Cassette cassette) {
  this.cassette = cassette;
}
```

```
@Autowired
public void insertCassette(Cassette cassette) {
```

```
  this.cassette = cassette;
}
```

- If there are no matching beans, Spring will throw an exception as the application context is created
- You can set the required attribute to false
  - Spring will then perform autowiring, but if there are no matching beans, it will leave the bean unwired
  - Gotcha! Unwired properties can easily lead to `NullPointerException`s...

```
@Autowired(required=false)
public Walkman(Cassette cassette) {
  this.cassette = cassette;
}
```

- If there are multiple matching beans, Spring will also throw an exception indicating ambiguity
- You can replace `@Autowired` with `@Inject` from CDI
  - In most common cases, these annotations are interchangeable

```
@Named
public class Walkman {
  //...
  @Inject
  public Walkman(Cassette cassette) {
    this.cassette = cassette;
  }
  //...
}
```

- A small unit test to check the automatic configuration

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=WalkmanConfig.class)
public class WalkmanTest {
  @Rule // see http://stefanbirkner.github.io/system-rules/index.html
  public final StandardOutputStreamLog log = new StandardOutputStreamLog();

  @Autowired
  private MediaPlayer player;

  @Autowired
  private Cassette cassette;

  //...

  @Test
  public void play() {
    player.play();
    assertEquals("Playing Awesome Mix Volume 1 by Various Artists\n", log.getLog());
  }
}
```

3.6. Wiring beans with Java

- There are times when you need to configure Spring explicitly
  - E.g. wiring components from third party libraries
- JavaConfig is the preferred way for explicit configuration
  - It is type-safe and refactor-friendly
  - But, JavaConfig is not just any Java code, it is configuration code
  - Avoid putting any business logic in Java configurations
  - Avoid doing any Java configuration in your business logic

- Use `@Configuration` to create a JavaConfig class
  - This annotation identifies it as a class containing bean details to be created in Spring

```
@Configuration
public class WalkmanConfig {
}
```

- We can replace the `@ComponentScan` with explicit configuration
  - There is no reason not to combine them, but serves as an example
  - The `@Bean` annotation tells Spring this method returns a bean that should be created and added to the application context
  - The name of the bean will be the name of the method

```
@Bean
public Cassette awesomeMix() {
  return new AwesomeMixVolume1();
}
```

- You can change the name of the bean with the name attribute

```
@Bean(name="guardiansOfTheGalaxySoundtrack")
public Cassette awesomeMix() {
  return new AwesomeMixVolume1();
}
```

- Since the configuration is in Java, you could do whatever you want to create a bean, as you have the whole Java power behind it
- The easiest way to wire beans together, is by using the referenced bean method

```
@Bean
public Walkman walkman() {
  return new Walkman(awesomeMix());
}
```

- While it looks like the `awesomeMix()` method will be called, this is not true
  - Because of `@Bean`, Spring will intercept the calls to the method, and return the bean produced by the method
  - By default, all Spring beans are singletons
- Another way to refer to a bean, is by using a parameter, with autowiring
  - This does not depend on the `Cassette` being configured in the same configuration class
  - Spring can also wire in the `Cassette` from XML or component scanning

```
@Bean
public Walkman walkman(Cassette cassette) {
  return new Walkman(cassette);
}
```

- Nothing stops you from using setters, as any Java code is allowed

```
@Bean
public Walkman walkman(Cassette cassette) {
  Walkman walkman = new Walkman();
  walkman.setCassette(cassette);
  return walkman;
}
```

3.7. Wiring with XML

- XML is not the only Spring configuration option
  - XML should not be your first choice, with JavaConfig and automatic configuration available
- XML was the primary way of expressing configuration
  - Lots of configuration files were already written for Spring
  - It is still important to understand how to use XML in existing applications

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans.xsd">
  <!--configuration details go here -->
</beans>
```

- Declare a bean with the `<bean>` element

```
<bean class="musicplayer.AwesomeMixVolume1" />
```

- It is a good `id`(ea) to give your bean a name for referring to it later
  - The default name of the bean would be `musicplayer.AwesomeMixVolume1#0`

```
<bean id="awesomeMix" class="musicplayer.AwesomeMixVolume1" />
```

- Spring will create the bean using its default constructor
  - The XML is a more passive configuration
  - XML is less powerful, as in JavaConfig you can do almost anything to arrive at a bean instance
  - The XML also does not have compile-time verification
- Constructor injection

```
<bean id="walkman" class="musicplayer.Walkman">
  <constructor-arg ref="awesomeMix" />
</bean>
```

- A shorter notation is provided from Spring 3 onwards

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
          xmlns:c="http://www.springframework.org/schema/c"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="awesomeMix" class="musicplayer.AwesomeMixVolume1" />
  <bean id="walkman" class="musicplayer.Walkman" c:cassette-ref="awesomeMix"/>
  <!--bean id="walkman" class="musicplayer.Walkman" c:_0-ref="awesomeMix" -->
  <!--bean id="walkman" class="musicplayer.Walkman" c:_-ref="awesomeMix"-->
</beans>
```

- Injecting literal values

```
public class BlankCassette implements Cassette {
  private String title;
  private String artist;

  public BlankCassette(String title, String artist) {
    this.title = title;
    this.artist = artist;
  }

  public void play() {
    System.out.println("Playing " + title + " by " + artist);
  }
}
```

```
<bean id="awesomeMix" class="musicplayer.BlankCassette">
  <constructor-arg value="Awesome Mix Volume 1" />
  <constructor-arg value="Various Artists" />
</bean>
```

- This can also be shortened out with the `c`-namespace attributes

```
<bean id="awesomeMix" class="musicplayer.BlankCassette" c:_title="Awesome Mix Volume 1"
  c:_artist="Various Artists"/>
```

- Or, using the numbered shorthands

```
<bean id="awesomeMix" class="musicplayer.BlankCassette" c:_0="Awesome Mix Volume 1"
  c:_1="Various Artists"/>
```

- The constructor-arg element lets you configure collections

```
public class BlankCassette implements Cassette {
  // ...
  private List<String> tracks;

  public BlankCassette(String title, String artist, List<String> tracks) {
    // ...
    this.tracks = tracks;
  }
  // ...
}
```

- You could configure it as empty, using the `null` element

```
<bean id="awesomeMix" class="musicplayer.BlankCassette">
  <constructor-arg value="Awesome Mix Volume 1" />
  <constructor-arg value="Various Artists" />
  <constructor-arg><null/></constructor-arg>
</bean>
```

```
<bean id="awesomeMix" class="musicplayer.BlankCassette">
  <constructor-arg value="Awesome Mix Volume 1" />
  <constructor-arg value="Various Artists" />
  <constructor-arg>
    <list> <!-- wiring a java.util.List -->
      <value>Hooked on a Feeling</value>
      <value>Go All the Way</value>
      <value>Spirit in the Sky</value>
      <value>Moonage Daydream</value>
      <value>Fooled Around and Fell in Love</value>
      <!-- ...other tracks omitted for brevity... -->
    </list>
  </constructor-arg>
</bean>
```

- Using `<ref>` you can reference other beans
- With `<set>` you can wire in a `java.util.Set`

- Setting properties
  - The `property` element will use the setter to wire in the dependency

```
<bean id="walkman" class="musicplayer.Walkman">
  <property name="cassette" ref="awesomeMix" />
</bean>
```

- Constructor injection or setter injection
  - Constructor injection implements hard dependencies
    - The bean cannot exist without its dependencies being satisfied

- Setter injection can denote optional dependencies
  - Imagine the bean being able to function without having the dependency satisfied
- Spring offers you the choice!

- A shorter notation with the `p`-namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="walkman" class="musicplayer.Walkman" p:cassette-ref="awesomeMix"/>
</bean>
```

- Setting values

```
<bean id="awesomeMix" class="musicplayer.BlankCassette">
  <property name="title" value="Awesome Mix Volume 1" />
  <property name="artist" value="Various Artists" />
  <property name="tracks">
    <list>
      <value>Hooked on a feeling</value>
      <!-- ...other tracks omitted for brevity... -->
    </list>
  </property>
</bean>
```

- There is no convenient way to wire collections with the `p`-namespace
  - So you can use the `util`-namespace instead

```
<?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:p="http://www.springframework.org/schema/p"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/util
  http://www.springframework.org/schema/util/spring-util.xsd">
  <!-- ... -->
</beans>
```

- The `util`-namespace offers the `<util:list>` element, that lets you create the list as a bean of its own

```
<util:list id="trackList">
  <value>Hooked on a Feeling</value>
  <value>Go All the Way</value>
  <value>Spirit in the Sky</value>
  <value>Moonage Daydream</value>
  <value>Fooled Around and Fell in Love</value>
  <!-- ...other tracks omitted for brevity... -->
</util:list>
```

- You can now use the `p`-namespace to declare the bean

```
<bean id="awesomeMix" class="musicplayer.BlankCassette"
  p:title="Awesome Mix Volume 1"
  p:artist="Various Artists"
  p:tracks-ref="trackList" />
```

- Other elements in the `util`-namespace

Element	Description

`<util:constant>`	References a public static field on a type and exposes it as a bean
`<util:list>`	Creates a bean that is a `java.util.List` of values or references
`<util:map>`	Creates a bean that is a `java.util.Map` of values or references
`<util:properties>`	Creates a bean that is a `java.util.Properties`
`<util:property-path>`	References a bean property (or nested property) and exposes it as a bean
`<util:set>`	Creates a bean that is a `java.util.Set` of values or references

3.8. Importing and mixing configurations

- None of the Spring configuration options are mutually exclusive
  - You are free to mix component scanning and autowiring with JavaConfig and/or XML
- Remember it does not matter where the bean comes from!
- Importing beans
  - The `@Import` annotation will let you bring the two configurations together

```
@Configuration
public class CassetteConfig{
  @Bean public Cassette cassette() {
    return new AwesomeMixVolume1();
  }
}
```

```
@Configuration
@Import(CassetteConfig.class)
public class WalkmanConfig{
  @Bean public Walkman walkman(Cassette cassette) {
    return new Walkman(cassette);
  }
}
```

- When the beans come from an XML file, you can import it with `@ImportResource` into a JavaConfig

```
@Configuration
@Import(CassetteConfig.class)
@ImportResource("classpath:cassette-config.xml")
public class WalkmanConfig{
}
```

- You can also break XML files apart, then import them in one another

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"...>
  <import resource="cassette-config.xml" />
  <bean id="walkman" class="musicplayer.Walkman" c:cassette-ref="cassette" />
</beans>
```

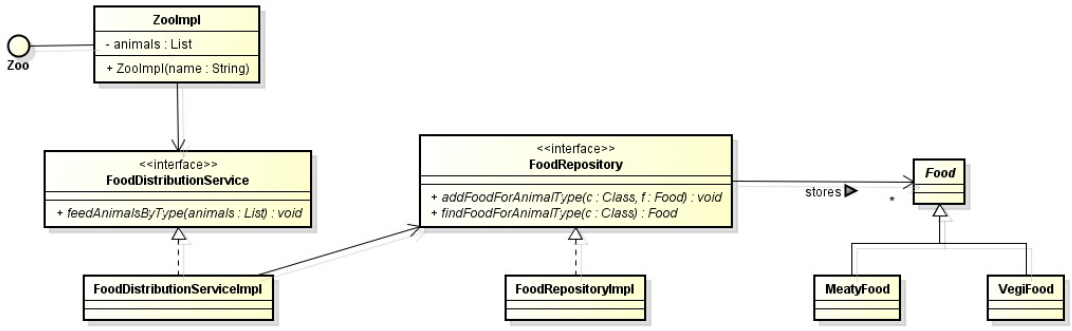
- To import a JavaConfig class in XML, you declare it as a bean in the XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"...>
  <bean class="musicplayer.CassetteConfig" />
  <bean id="walkman" class="musicplayer.Walkman" c:cassette-ref="cassette"/>
</beans>
```

- Best practice
  - Create a root configuration that brings JavaConfig and XML files together (be it in XML or Java)
  - Activate component scanning in this root configuration for automatic configuration

3.9. Exercise: Wiring

- Add a `FoodRepository` interface and implementation
  - `addFoodForAnimalType(Class<? extends Animal> clazz, Food food)`
  - `findFoodForAnimalType(Class<? extends Animal> clazz)`
- Add a `FoodDistributionService` interface and implementation
  - `feedAnimalsByType(List<Animal> animals)`
- Add the following method to the Zoo
  - `feedAnimals()`
- Wire the `FoodRepository` to the `FoodDistributionService`
- Wire the `FoodDistributionService` to the `ZooImpl`
- Note: Configure the wiring in any way you like, Java or XML or both!



4. Wiring Beans Advanced

4.1. Environments and profiles

- Transitioning an application from one environment to another is one of the most challenging aspects in software development
  - From development to production, many things can change: database configuration, encryption algorithms, integration and external systems, ...
- Consider following `DataSource`
  - This `DataSource` is useful in development, but horrible in production!

```
@Bean(destroyMethod="shutdown")
public DataSource dataSource() {
  return new EmbeddedDatabaseBuilder()
    .setType(EmbeddedDatabaseType.H2)
    .addScript("classpath:schema.sql")
    .addScript("classpath:test-data.sql")
    .build();
}
```

- In production we would rather use a `DataSource` from our container using JNDI
  - This is more fitting for production, but unnecessarily complicated for a simple developer test

```
@Bean
public DataSource dataSource() {
  JndiObjectFactoryBean jndiObjectFactoryBean = new JndiObjectFactoryBean();
  jndiObjectFactoryBean.setJndiName("jdbc/myDS");
  jndiObjectFactoryBean.setResourceRef(true);
  jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
  return (DataSource) jndiObjectFactoryBean.getObject();
}
```

- And for QA testing, you would perhaps prefer another `DataSource` created with a Commons DBCP connection

```
@Bean(destroyMethod="close")
public DataSource dataSource() {
  BasicDataSource dataSource = new BasicDataSource();
  dataSource.setUrl("jdbc:h2:tcp://dbserver/~/test");
  dataSource.setDriverClassName("org.h2.Driver");
  dataSource.setUsername("sa");
  dataSource.setPassword("password");
  dataSource.setInitialSize(20);
  dataSource.setMaxActive(30);
  return dataSource;
}
```

- All three configurations are different, but valid
  - We need a way to chose the most appropriate configuration for each environment
- Spring introduces bean profiles
  - You can configure beans into one or more profiles
  - At runtime, Spring makes the decision which beans need to be created
  - The same deployment unit will work in all environments without being rebuilt
- Use the `@Profile` annotation

```
@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();
  }
}
```

- Another JavaConfig may have another profile
  - The bean will not be created unless the corresponding profile is active
  - Any bean with no profile will always be created, regardless which profile is active

```
@Configuration
@Profile("prod")
public class ProductionProfileConfig{
  @Bean
  public DataSource dataSource() {
    JndiObjectFactoryBean jndiObjectFactoryBean= new JndiObjectFactoryBean();
    jndiObjectFactoryBean.setJndiName("jdbc/myDS");
    jndiObjectFactoryBean.setResourceRef(true);
    jndiObjectFactoryBean.setProxyInterface(
    javax.sql.DataSource.class);
    return (DataSource) jndiObjectFactoryBean.getObject();
  }
}
```

- You can add the profile on the method level
  - This makes it possible to combine both bean declarations into a single configuration

```
@Configuration
public class DataSourceConfig{
  @Bean(destroyMethod="shutdown")
  @Profile("dev")
  public DataSource embeddedDataSource() {
    return new EmbeddedDatabaseBuilder()
    //...
    .build();
  }
  @Bean
  @Profile("prod")
  public DataSource jndiDataSource() {
    JndiObjectFactoryBean jndiObjectFactoryBean = new JndiObjectFactoryBean();
    //...
    return (DataSource) jndiObjectFactoryBean.getObject();
  }
}
```

- To activate a profile, you can use two properties
  - `spring.profiles.active`
  - `spring.profiles.default`
- These values can be set in several ways
  - As initialization parameters on `DispatcherServlet`
  - As context parameters of a web application
  - As JNDI entries
  - As environment variables
  - As JVM system properties
  - Using the `@ActiveProfiles` annotation on an integration test class
- A possible scenario
  - Set the `spring.profiles.default` on `"dev"`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" ...>
  <!-- ... -->
  <context-param>
    <param-name>spring.profiles.default</param-name>
    <param-value>dev</param-value>
  </context-param>
```

```
  <!-- ... -->
  <servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>spring.profiles.default</param-name>
      <param-value>dev</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
</web-app>
```

- A possible scenario
  - When the application is deployed to QA or production, set `spring.profiles.active` using system properties, environment variables, JNDI or other
  - These will then take precedence in the corresponding environment
- You can make several profiles active at the same time by listing the profile names, separated by commas
- In testing, you can use the `@ActiveProfiles` annotation

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={PersistenceTestConfig.class})
@ActiveProfiles("dev")
public class PersistenceTest {
  // ...
}
```

4.2. Conditional beans

- Suppose
  - You only want a bean if and only if some library is available
  - Or if a certain other bean is also declared
  - Or if a specific environment variable is set
- Spring 4 introduces a new `@Conditional` annotation you can add on `@Bean` methods
  - If the condition evaluates to true, the bean is created, otherwise it is ignored
  - The `@Profile` annotation has been refactored to be based on `@Conditional`

```
@Bean
@Conditional(SmurfsExistsCondition.class)
public MagicBean magicBean() {
  return new MagicBean();
}
```

- The `@Conditional` comes with a `Condition` interface

```
public interface Condition {
  boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);
}
```

- Example, you could check for the existence of a property in the environment

```
public class SmurfsExistsCondition implements Condition {
  public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
    Environment env = context.getEnvironment();
    return env.containsProperty("smurfs");
  }
}
```

- There are much more that can be considered, from the `ConditionContext` and `AnnotatedTypeMetadata` interfaces

4.3. Addressing ambiguity in autowiring

- Autowiring only works when exactly one bean matches the desired result
  - When there is more than one bean, ambiguity prevents Spring from autowiring
- Illustration

```
@Autowired
public void setDessert(Dessert dessert) {
  this.dessert = dessert;
}
```

```
@Component
public class Cake implements Dessert { ... }
```

```
@Component
public class Cookies implements Dessert { ... }
```

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

- When Spring is unable to choose, it will fail with an exception
  - `NoUniqueBeanDefinitionException`

```
nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [com.desserteater.Dessert] is defined: expected single matching bean but found 3:
cake,cookies,iceCream
```

- Remark
  - In reality, autowiring ambiguity is rare
  - More often than not, there is only one implementation of a given type
- Solution
  - Designate one bean as primary choice, or use qualifiers
- Using `@Primary`, Spring will choose this bean over any other candidate beans

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

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

- But you could also designate more than one primary bean... which does not solve the problem...
- Use `@Qualifier` to narrow down to a single bean
  - If ambiguity still exists after applying all qualifiers, you can always add more to narrow the choices further

```
@Autowired
@Qualifier("iceCream")
public void setDessert(Dessert dessert) {
  this.dessert = dessert;
}
```

- The parameter given to qualifier is the id of the bean

- By default, all beans are given a qualifier that is the same as their bean id
- This can be problematic when refactoring however...

- Instead of relying on the bean id, you can assign your own qualifier
  - It is recommended to use a trait or descriptive term for the bean, rather than an arbitrary name

```
@Component
@Qualifier("cold")
public class IceCream implements Dessert { ... }
```

```
@Autowired
@Qualifier("cold")
public void setDessert(Dessert dessert) {
  this.dessert = dessert;
}
```

- But using qualifiers with traits can still get you into trouble

```
@Component
@Qualifier("cold")
public class Popsicle implements Dessert { ... }
```

- You can then use additional qualifiers to narrow down the selection
  - But... this is not allowed in Java however... and `@Qualifier` is not `@Repeatable`

```
@Component
@Qualifier("cold")
@Qualifier("creamy")
public class IceCream implements Dessert { ... }
```

```
@Autowired
@Qualifier("cold")
@Qualifier("creamy")
public void setDessert(Dessert dessert) {
  this.dessert = dessert;
}
```

- You can then create your own qualifier annotations

```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD, ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Cold { }
```

```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD, ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Creamy { }
```

```
@Component
@Cold
@Creamy
public class IceCream implements Dessert { ... }
```

```
@Autowired
@Cold
@Creamy
public void setDessert(Dessert dessert) { this.dessert = dessert; }
```

4.4. Scoping beans

- By default, all beans are created as singletons
  - This is ideal most of the time: the cost of instantiating and garbage collection cannot be justified for stateless objects that can be reused over and over again
- But sometimes, you have classes with mutable state, that are not safe for reuse...
- Spring defines several scopes under which a bean can be created
  - Singleton—One instance for the entire application
  - Prototype—One instance every time the bean is injected or retrieved
  - Session—One instance for each session in a web application
  - Request—One instance for each request in a web application
- To select an alternative scope, use the `@Scope` annotation
- Setting the prototype scope

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class Notepad { ... }
```

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public Notepad notepad() {
  return new Notepad();
}
```

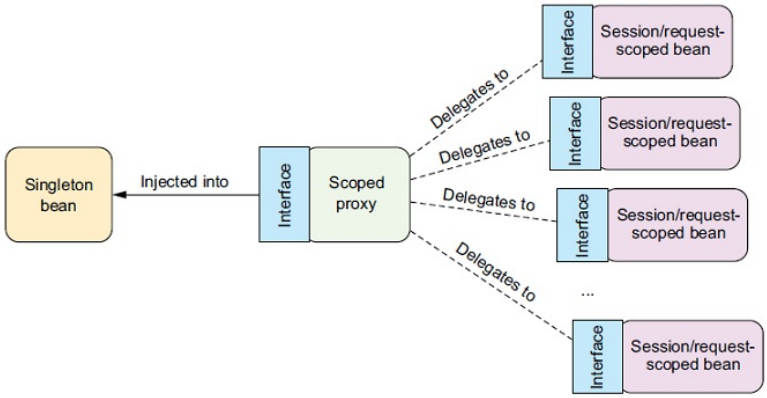
- An instance of the bean will be created each and every time it is injected into or retrieved from the Spring application context
- In web applications, it is useful to instantiate a bean in session scope
  - E.g. a shopping cart

```
@Bean
@Scope(value=WebApplicationContext.SCOPE_SESSION,
  proxyMode=ScopedProxyMode.INTERFACES)
public ShoppingCart cart() { ... }
```

  - The `proxyMode` attribute is required in this case
  - This allows the reference to resolve to the actual session-scoped shopping cart

```
@Component // singleton
public class StoreService{
  @Autowired // which one? not a singleton, but a proxy
  public void setShoppingCart(ShoppingCart shoppingCart) {
    this.shoppingCart = shoppingCart;
  }
  // ...
}
```

- Setting the `proxyMode` to `ScopedProxyMode.INTERFACES`
  - This indicates that the proxy should implement the `ShoppingCart` interface and delegate to the implementation bean
- Setting the `proxyMode` to `ScopedProxyMode.TARGET_CLASS`
  - This indicates that the proxy should be generated as subclass of the target class



4.5. Runtime value injection

- Wiring is not only about object references
  - Sometimes you want to inject values into a bean

```
@Bean
public Cassette awesomeMix() {
  return new BlankCassette("Awesome Mix Volume 1", "Various Artists");
}
```

- This will hardcode the values into the configuration class
- Spring offers two solutions to evaluate values at runtime
  - Property placeholders
  - The Spring Expression Language (SpEL)
- The simplest way to inject external values is by declaring a property source

```
@Configuration
@PropertySource("classpath:/com/musicplayer/app.properties")
public class ExpressiveConfig{
  @Autowired
  Environment env;

  @Bean
  public BlackCassette cassette() {
    return new BlankCassette(env.getProperty("cass.title"), env.getProperty("cass.artist"));
  }
}
```

- The `@PropertySource` references a file with properties

```
cass.title=Awesome Mix Volume 1
cass.artist=Various Artists
```

- The `Environment` has overloaded `getProperty()` methods
  - `String getProperty(String key)`
  - `String getProperty(String key, String defaultValue)`
  - `T getProperty(String key, Class<T> type)`
  - `T getProperty(String key, Class<T> type, T defaultValue)`
- You can then specify a default value if the properties do not exist

```
@Bean
public BlankCassette cassette() {
  return new BlankCassette(env.getProperty("cass.title", "The Very Best Of"),
    env.getProperty("cass.artist", "Bob Marley"));
```

```
}
```

- If the property is required, you can also use `getRequiredProperty()`
- From the `Environment` you can retrieve the active profiles
- Properties can be injected using property placeholders in code
  - Use `@Value` in the same way as `@Autowired` but for values

```
public BlankCassette(@Value("${cass.title}") String title, @Value("${cass.artist}") String artist) {
  this.title = title;
  this.artist = artist;
}
```

- This requires a `PropertySourcesPlaceholderConfigurer`
  - This bean will resolve using the Spring Environment and its set of property sources

```
@Bean
public static PropertySourcesPlaceholderConfigurer placeholderConfigurer() {
  return new PropertySourcesPlaceholderConfigurer();
}
```

4.6. Wiring with Spring Expression Language

- SpEL is a powerful but compact way of wiring values into bean properties or arguments using expressions that are evaluated at runtime
  - Use `#{...}` instead of `${...}`
- SpEL has lots of capabilities
  - Reference beans by their identifiers
  - Invoking methods and accessing properties on objects
  - Mathematical, relational, and logical operations on values
  - Regular expression matching
  - Collection manipulation
- SpEL can also be used in combination with Thymeleaf, MVC, Security, Web Flow and more...
- Examples

```
#{1} // returns 1
```

```
#{T(System).currentTimeMillis()} // working with Types
```

```
#{awesomeMix.artist} // refer to bean properties
```

```
#{systemProperties['cass.title']} // refer to system properties
```

```
#{3.14159} #{'Hello'} #{false} // literals
```

```
#{artistSelector.selectArtist().toUpperCase()} // calling bean methods
```

```
#{artistSelector.selectArtist()?.toUpperCase()} // type-safe operator, protects against null
```

- You can also use operators in SpEL

Operator type	Operators

Arithmetic	`+`, `-`, `*`, `/`, `%`, `^`
Comparison	`<`, `lt`, `>`, `gt`, `==`, `eq`, `<=`, `le`, `>=`, `ge`
Logical	`and`, `or`, `not`
Conditional	`?:` (ternary), `?:` (Elvis)
Regularexpression	`matches`

- Examples

```
#{T(java.lang.Math).PI * circle.radius ^ 2} // a circle's area
```

```
#{cass.title + ' by ' + cass.artist} // String concatenation
```

```
#{scoreboard.score > 1000 ? "Winner!" : "Loser"} // ternary operator
```

```
#{cass.title ?: 'The Very Best of'} // Elvis operator
```

```
#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.com'} // regular expression
```

- A few tricks of SpEL on collections and arrays

```
#{jukebox.songs[4].title} // referencing a value from an array or list
```

```
#{jukebox.songs[T(java.lang.Math).random() * jukebox.songs.size()].title} // random select
```

```
#{jukebox.songs.?[artist eq 'Bob Marley']} // filter the collection on the artist
```

```
#{jukebox.songs.^[artist eq 'Bob Marley']} // selecting the first occurrence
```

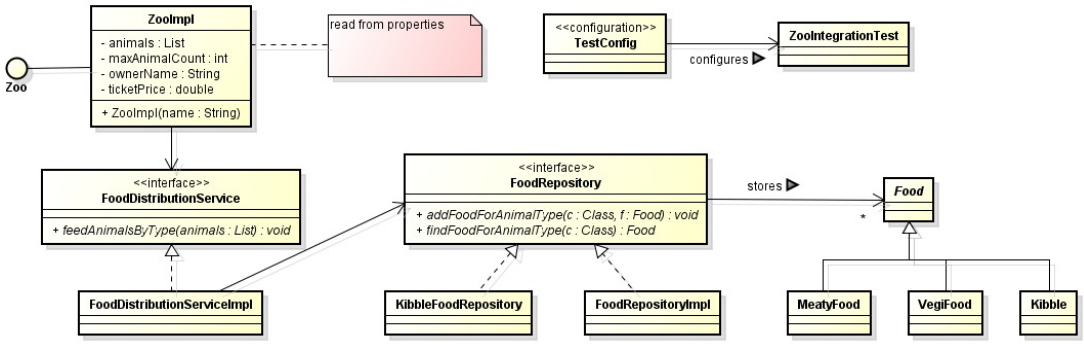
```
#{jukebox.songs.$[artist eq 'Bob Marley']} // selecting the last occurrence
```

```
#{jukebox.songs.![title]} // projection on the titles
```

- Warning: do not be too clever with SpEL expressions, because they can become difficult to test...

4.7. Exercise: Wiring Beans Advanced

- Add the following properties to the `ZooImpl`
  - `maxAnimalCount`
  - `ownerName`
  - `ticketPrice`
- Configure these properties and the Zoo `name` by using a properties file
- Create an alternate configuration for the Zoo, for use in testing
  - Use default testing values for the Zoo properties
- Use profiles to select the testing configuration in a JUnit test
- Create an alternate `FoodRepository`, giving all animals special "kibble"
- Select this new `FoodRepository` using your own `Qualifier`


ZooImpl
- animals : List
- maxAnimalCount : int
- ownerName : String
- ticketPrice : double
+ ZooImpl(name : String)
Zoo
read from properties
<<configuration>>
TestConfig
configures
ZooIntegrationTest
<<interface>>
FoodDistributionService
+ feedAnimalsByType(animals : List) : void
<<interface>>
FoodRepository
+ addFoodForAnimalType(c : Class, f : Food) : void
+ findFoodForAnimalType(c : Class) : Food
stores
*
Food
FoodDistributionServiceImpl
KibbleFoodRepository
FoodRepositoryImpl
MeatyFood
VegiFood
Kibble

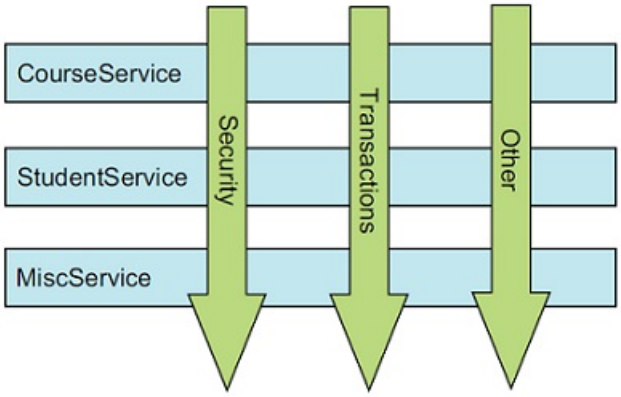
5. Aspect Orientation

5.1. Cross-cutting concerns

- In software development, some functions need to be invoked in multiple places, but it is undesirable to invoke them explicitly every time
  - Examples: logging, security, transaction management, caching, ...
- These are called "cross-cutting concerns"
- To keep the software manageable, cross-cutting concerns are often put in different compilation units
  - The Aspect Oriented Programming (AOP) allows us to do this
    - Allows us to decouple cross-cutting concerns from the business objects they affect

5.2. Aspect Oriented Programming

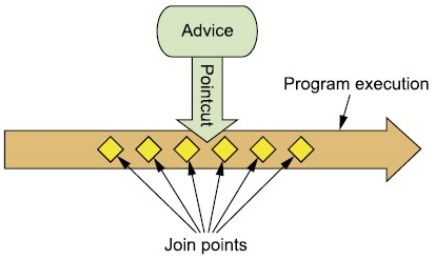
- Cross-cutting concerns are needed across several regular business services



- AOP allows us to modularize them as "aspects"
  - Offers a cleaner alternative to OO principles such as inheritance or delegation
  - Centralizes the logic concerned in a single location, without having a dependency to them in each business service

5.3. AOP terminology

- Just like OO, the AOP paradigm has its own jargon
- Aspects are described in terms of:
  - "advices", "join points" and "pointcuts"
- We say that:
  - "Advices are woven into the program's execution at specific join points selected by a pointcut"



5.4. Advices

- An advice represents an aspect's purpose
  - It controls two things: "what" to do and "when" to do it
- What:
  - The aspect's functionality or "job", as a fragment of (Java) programming logic
- When:

- An aspect is triggered upon an event in the program execution

Advice	Description
`@Before`	Functionality should trigger before execution of a business method
`@After`, `@AfterReturning`, `@AfterThrowing`	Functionality should trigger after execution of a business method
`@Around`	The advice is wrapped around the business method (before and after)

5.5. Join points

- A join point is a point in the normal program execution where an aspect can be plugged in
- Can be a variety of things
  - A method call
  - An exception being thrown
  - An object being instantiated
  - A field being assigned
- Your aspects will be inserted alongside these join points to "enhance" the normal business logic

5.6. Pointcuts

- Aspects can be assigned to join points selectively
  - Not all aspects need to be inserted on all join points
- A pointcut allows selection of "where" to plug an aspect in
  - In other words: "on which join points"
- Often this is done by using some regular expression like syntax to select one or more join points for inclusion

5.7. Aspects

- An aspect is simply the combination of a pointcut and an advice
  - This defines everything there is to know about: "what", "when" and "where"
- The name "aspect" allows us to talk about the concept as we do in natural language
  - For example:
    - "The security aspect of the system"
    - "Our application has many aspects to consider"
    - "Logging is only one aspect we need to deal with"

5.8. Introduction

- An introduction allows us to dynamically add new methods or state to a previously defined business class
- Since we want to do something with this new method, we need to know about it at compile-time
  - For this reason, introductions are usually done by dynamically implementing an additional interface on an existing class
    - The interface can be typed against in client logic

5.9. Weaving

- Weaving is the process of wrapping a proxy around a target object, so that the advices can intercept selected join points
  - A target object is a Java object that has a join point which is selected as the subject of an aspect
- Depending on the technology used, weaving can be done in multiple ways
  - At compile time
    - Requires a special compiler (AspectJ) that weaves aspects in using special syntax (not Java)
  - At class load time
    - Uses a special `ClassLoader` to weave aspects into a class when it's being loaded in the JVM
  - At runtime
    - Dynamically weaves in aspects at runtime by using a container or engine (like Spring or Java EE)

5.10. AOP frameworks

- There are many implementations of AOP systems
  - They differ in the join points they support and their weaving model
- JBoss AOP
  - Supports more join points than Spring AOP
- AspectJ
  - Supports both runtime and compile time weaving
  - Extends Java syntax, but required more complex build configuration
  - Very fine grained AOP support
- Spring AOP

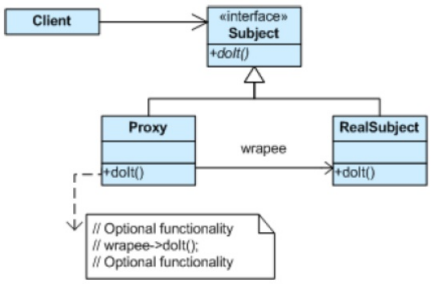
5.11. Spring AOP

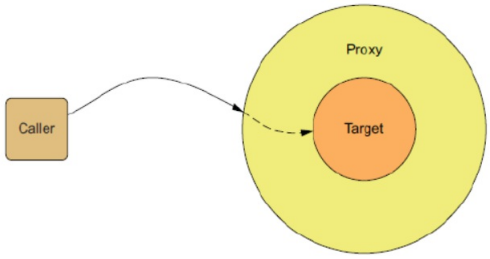
- Spring's AOP engine can be used in multiple ways
  - Using annotations (preferred)
    - By borrowing the AspectJ runtime annotations, you can configure aspects on business objects by adding a few simple annotations
  - Using XML and pure POJOs
    - Using only XML configuration, you can configure aspects for business objects without modifying their source files
  - Low level using interfaces and classes to weave proxies around target objects
    - The outdated, but internally still used to implement the previous two
  - Using AspectJ compiled classes
    - The most powerful solution, requiring you to program aspects in AspectJ's special language which is then compiled to JVM compatible bytecode
- Spring uses Java to write your advices
  - Allows you to reuse all existing Java knowledge at the price of some expressiveness
- Spring uses runtime weaving
  - Target objects will be proxied using the proxy design pattern to allow applying advices before, after or around business methods
  - These proxy objects are lazily created at runtime by the `ApplicationContext`
- Spring supports only method join points

- Suffices for most scenarios, while keeping the complexity in check

5.12. AOP proxies

- Spring AOP proxies are based on the proxy design pattern





5.13. Configuring aspects using annotations

- Annotations is the preferred way to configure aspects in Spring 4
  - We will primarily focus on this style
- Spring borrows the AspectJ annotations and expression language for this
  - Only supports a subset of the full set of AspectJ designators
- The full documentation on AspectJ can be found here:
  - https://eclipse.org/aspectj/
- We will explore the most useful expressions on the next couple of slides

5.14. Dependencies

- Since Spring is a modular framework, we need to add the AOP module as a dependency to the project
- When using Maven, this can be done by adding the following dependencies
  - `org.springframework:spring-aop`
  - `org.aspectj:aspectjrt`
  - `org.aspectj:aspectjweaver`
- When using Spring Boot, this is reduced to a single dependency

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

5.15. Enabling AOP

- To enable the Spring AOP engine, the `ApplicationContext` configuration must be annotated with `@EnableAspectJAutoProxy`

```
@Configuration
@ComponentScan
@EnableAspectJAutoProxy
public class MyConfiguration {
  @Bean
```

```
  public Performer performer() {
    return new Concert(...);
  }
}
```

5.16. Pointcut designators

- Pointcuts use designators to select a range of joinpoints on which their advice must act
  - Most of them have some sort of regexp-style syntax
- Each of these designators has its own specific expression syntax
  - We will explore this syntax by taking some examples from the book and the Spring reference manual
- Each expression can be chained using the logical operators or their word equivalents
  - `&&` (`and`), `||` (`or`), `!` (`not`)
- Here is a list of AspectJ expressions supported by Spring

Designator	Description
`args()`, `@args()`	Limits join-point matches to the execution of methods whose arguments are instances of the given types, or are annotated with the given annotation types.
`execution()`	Matches join points that are method executions.
`this()`	Limits join-point matches to those where the bean reference of the AOP proxy is of a given type.
`target()`, `@target()`	Limits join-point matches to those where the target object is of a given type or is annotated with the given annotation type. Matched at runtime and can be used to bind the target type.
`within()`, `@within()`	Static version of target() and @target(), which is faster but does not support binding.
`@annotation()`	Limits join-point matches to those where the subject of the join point has the given annotation.

5.17. Execution

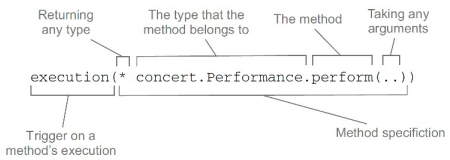
- The `execution` designator uses the following syntax

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)
```

- You can use the `*` wildcard anywhere

Pattern	Explanation
modifiers-pattern	Visibility modifiers such as "public". Optional.
ret-type-pattern	FQN for the return type. Required.
declaring-type-pattern	FQN of the class that owns the joinpoint. Optional.
name-pattern	The joinpoint (method) to intercept. Required.
param-pattern	Comma separated list of FQN parameter names. May be empty.
throws-pattern	FQN of the throwing exception. Optional.

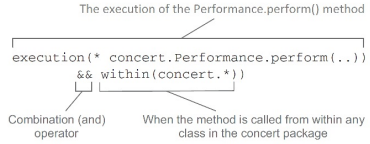
- Some examples using the Performance interface




```
package concert;

public interface Performance {
  public String perform();
}
```

- Some examples using the Performance interface

```
package concert;

public class Concert implements Performance {
  public String perform() {
    System.out.println("Keep on rocking in the free world!");
    return "Neil Young";
  }
}
```



- The execution of any public method

```
execution(public * *(..))
```

- The execution of any method with a name beginning with "set"

```
execution(* set*(..))
```

- The execution of any method defined by the `AccountService` interface

```
execution(* com.xyz.service.AccountService.*(..))
```

- The execution of any method defined in the `service` package

```
execution(* com.xyz.service.*.*(..))
```

- The execution of any method defined in the `service`- or sub-package

```
execution(* com.xyz.service..*.*(..))
```

5.18. Within, this and target

- Any method `within` the service package

```
within(com.xyz.service.*)
```

- Any method within the service package or a sub-package

```
within(com.xyz.service..*)
```

- Any method where the type of the target object has an `@Transactional` annotation

```
@within(org.springframework.transaction.annotation.Transactional)
```

- Any method where the proxy implements the `AccountService` interface

```
this(com.xyz.service.AccountService)
```

- Any method where the target object implements the `AccountService` interface

```
target(com.xyz.service.AccountService)
```

- Any method where the target object has an `@Transactional` annotation:

```
@target(org.springframework.transaction.annotation.Transactional)
```

5.19. Args

- Any method which takes a single parameter, and where the runtime type of the argument passed has the `@Classified` annotation

```
@args(com.xyz.security.Classified)
```

- Any method which takes a single parameter, and where the argument passed at runtime is `Serializable`

```
args(java.io.Serializable)
```

5.20. Annotations

- Any method where the executing method has an `@Transactional` annotation

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

5.21. Bean

- Any method on a Spring bean named "tradeService"

```
bean(tradeService)
```

- Any method on Spring beans having names that match the wildcard expression `*Service`

```
bean(*Service)
```

- Only the `perform()` method on instances of `Performance` that are known in the Spring context with name "woodstock"

```
execution(* concert.Performance.perform()) and bean(woodstock)
```

5.22. Aspect beans

- Aspects are created in Spring AOP as regular Java classes
  - These classes need to be registered in the `ApplicationContext` using any of the available methods
- Using `@Bean` and `@Configuration`

```
@Bean public Audience audience() { return new Audience(); }
```

- Or using `@Component` and `@ComponentScan`

```
@Component public class Audience { }
```

- Or using XML and `<bean>` elements
- On top of that, they need to be marked as an aspect using the `@Aspect` annotation

```
@Component // <--in case you chose the component scanning method
@Aspect
public class Audience {
  // Advices will be put inside the aspect
}
```

- Advices are created by adding a method to an aspect bean that is marked with an Advice annotation
  - The method body will control the "what"
  - The annotation type will control the "when"
  - The embedded pointcut expression will control the "where"

```
@Aspect
public class Audience {
  @Before("execution(* concert.Performance.perform(..))")
  public void takeSeats() { System.out.println("Taking seats"); }
}
```

- There are five advice types

Annotation	Advice
`@Before`	The advice method is called before the advised method.
`@After`	The advice method is called after the advised method.
`@AfterReturning`	The advice method is called only after the advised method returns gracefully by returning normally.
`@AfterThrowing`	The advice method is called only after the advised method returns ungracefully by throwing an exception.
`@Around`	The advice method is called before and after the advised method. This is the most powerful advice type, but also the most dangerous.

5.23. Before advice

- The `@Before` advice is the simplest of the advice types

```
@Before("execution(* concert.Performance.perform(..))")
public void silenceCellPhones() {
  System.out.println("Silencing cell phones");
}

@Before("execution(* concert.Performance.perform(..))")
public void takeSeats() {
  System.out.println("Taking seats");
}
```

- It allows the advised method to be intercepted before being called itself
  - This can be used to "prepare" context for the advised method, such as opening a connection to a resource, so that the advised method doesn't need to worry about it anymore

5.24. After advice

- The `@After` advice has three shapes
  - `@After` (any), `@AfterReturning` and `@AfterThrowing`

```
@AfterReturning("execution(* concert.Performance.perform(..))")
public void applause() {
  System.out.println("CLAP CLAP CLAP!!!");
}

@AfterThrowing("execution(* concert.Performance.perform(..))")
public void demandRefund() {
  System.out.println("Boo! Demanding a refund.");
}
```

- It allows the advised method to be "post-processed" depending on the exit condition

5.25. After advice capturing

- `@AfterReturning` and `@AfterThrowing` are triggered after the advised method has been called
  - This means the advised method's return or exception values are already "known"
- These returned / thrown object can be captured by the advice

```
@AfterReturning(value="execution(* concert.Performance.perform(..))", returning="artist")
public void returnCapturingAdvice(String artist) {
  System.out.println("Great performance by" + artist);
}

@AfterThrowing(value="execution(* concert.Performance.perform(..))", throwing="myException")
public void exceptionCapturingAdvice(IllegalArgumentException myException) {
  System.out.println("Bad concert due to" + myException.getMessage());
}
```

5.26. Around advice

- The `@Around` advice is somewhat special, because it is very powerful
  - It wraps the advised method completely, effectively combining a `@Before` and `@After`
- Because of this, it can exert full control over the advised method
  - Control whether or not the advised method is invoked at all
  - Modify or change the return value
  - Modify or change the thrown value
- This can be used for advanced cross-cutting concerns
  - Security: controlling access to an advised method
  - Transactions: opening before and commit/rollback after the advised method
- An `@Around` advice is used like this
  - They must receive an instance of `ProceedingJoinPoint` as their first parameter
    - Use it to control when and if to call the advised method
  - If the advised method returns or throws something you wish to propagate, you must also return or throw it, otherwise it will be "swallowed"

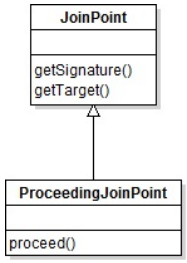
```
@Around("execution(* concert.Performance.perform(..))")
public String watchPerformance(ProceedingJoinPoint jp) throws Throwable{
  String artist = null;
  try{
```

```
    System.out.println("Silencing cell phones");
    artist = jp.proceed();
    System.out.println("CLAP CLAP CLAP!!!");
  } catch (Throwable e) {
    System.out.println("Demanding a refund");
  }
  return artist;
}
```

5.27. JoinPoint reflection

- Any advice can introspect the `JoinPoint` that it is triggered by
  - Simply add a parameter of type `JoinPoint` as the first of the advice method
  - This reference contains reflection-like introspection methods such as `getSignature()`

```
@Before("execution(* concert.Performance.perform(..))")
public void beforeAdvice(JoinPoint jp) {
  System.out.println("Called from" + jp.getSignature());
}
@AfterReturning(pointcut="execution(* concert.Performance.perform(..))",
returning="artist")
public void afterAdvice(JoinPoint jp, String artist) {
  System.out.println("Called on " + jp.getTarget());
}
@Around("execution(* concert.Performance.perform(..))")
public Object aroundAdvice(ProceedingJoinPoint jp) throwsThrowable{
  return jp.proceed();
}
```



5.28. Named pointcuts

- Previously, we have used the same pointcut expression multiple times
  - This is not efficient: a modification of this pointcut would require multiple changes
- To reuse pointcut expressions, you can use the `@Pointcut` annotation on an empty method
  - This results in a reusable pointcut named `performance()`

```
@Pointcut("execution(* concert.Performance.perform(..))") // pointcut expression
public void performance() {} // pointcut name
```

- You can use them in advice annotations

```
@Before("performance()")
public void silenceCellPhones() {
  System.out.println("Silencing cell phones");
}
```

5.29. Argument capturing

- All advice types can capture and bind the parameters from the advised method to a parameter of the advice method
  - Offers a similar feature as the returning and throwing attribute of the `@After`X advices
- Binding advised parameters is done using the `args()` designator
  - The name x of the advice method parameter must match the name used in `args(x)`

```
public class CompactDisk{
```

```
  public void playTrack(int trackNr) { ... } // <--The method being adviced
}
```

```
@Before("execution(* CompactDisk.playTrack(int)) && args(trackNumber)")
public void countTrack(int trackNumber) {
  System.out.println("Logging track played: " + trackNumber);
}
```

5.30. Introductions

- Java does not have the concept of an "open class"
  - These are classes that allow new methods to be (dynamically) added without modifying the class itself
    - Other languages do have this: Groovy, C#, Ruby, JavaScript ...
- The AOP concept of an "Introduction" allows us to emulate such a feature
  - We know that Spring implements AOP by creating a proxy that implements the same interface as the targeted class, and that this happens dynamically at runtime
  - When we let this proxy also implement another interface simultaneously, we have essentially emulated an "open class"
- The process of introducing behavior to existing classes is called a "mix-in"
  - They can be seen as a dynamic version of a trait or multiple-inheritance

```
public interface Performance {
  void perform();
}
```

```
public class Concert implements Performance {
  @Override
  public void perform() {
    System.out.println("Performing");
  }
}
```

```
public interface Encoreable{
  void performEncore();
}
```

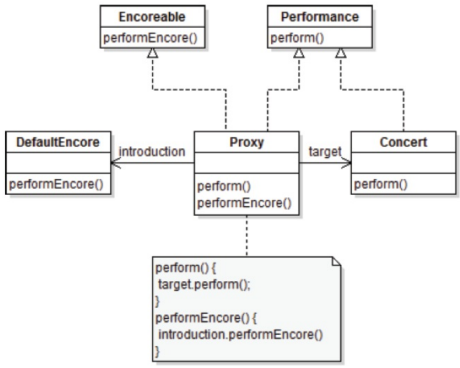
```
public class DefaultEncore implements Encoreable{
  @Override
  public void performEncore() {
    System.out.println("Encoring");
  }
}
```

- The previous classes can be mixed together using an introduction aspect

```
@Aspect
public class EncoreableIntroducer {
    @DeclareParents(value = "concert.Performance+", defaultImpl = DefaultEncore.class)
    public Encoreable encoreable;
}
```

The `+` sign here is not a typo. It specifies any subtype of `Performance`, as opposed to `Performance` itself.

- The configuration, usage and proxy structure looks like this




```
@Bean
public Concert performance() {
  return new Concert();
}
@Bean
public EncoreableIntroducer aspect() {
  return new EncoreableIntroducer();
}
```

```
Encoreable e = context.getBean("performance", Encoreable.class);
e.performEncore();

Performance p = context.getBean("performance", Performance.class);
p.perform();
```

5.31. XML configuration

- The preferred way to configure AOP in Spring 4 is using annotations
- If desired you can also configure everything using only XML
  - This way, your classes become pure POJOs (if you consider classes "polluted" with annotations as not being POJOs)
- For this, the `aop`-namespace is available

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">
    <!--bean definitions here-->
</beans>
```

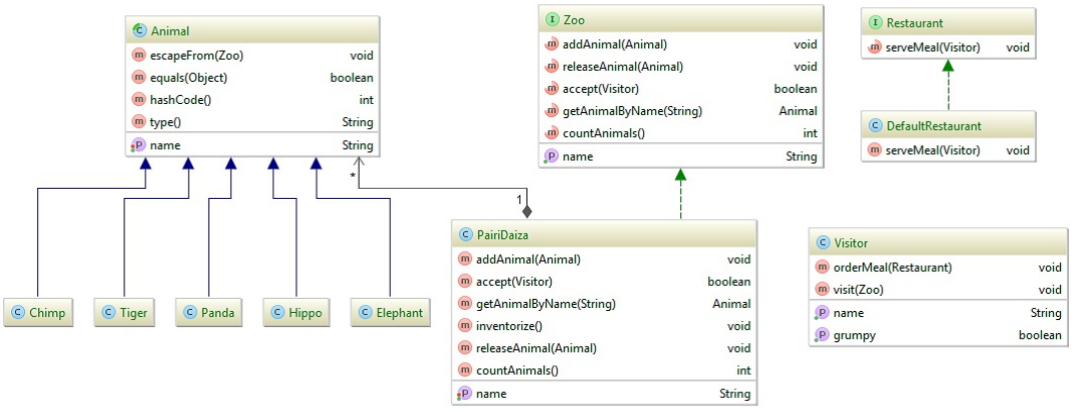
- You can use XML configuration that is fully equivalent to the annotations
  - All annotations can be removed from the source files

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut id="performance"
      expression="execution(* concert.Performance.perform(..))" />
    <aop:before pointcut-ref="performance" method="silenceCellPhones"/>
    <aop:after-returning pointcut-ref="performance" method="applause"/>
    <aop:after-throwing pointcut="execution(* concert.Performance.perform(..))"
      method="demandRefund" throwing="myException"/>
  </aop:aspect>
</aop:config>
<bean id="audience" class="concert.Audience"/>
```

```
<bean id="concert" class="concert.Concert"/>
```

5.32. Exercise: Aspect Orientation

- Open the "aspects" exercise provided by the Teacher
  - Explore the project to familiarize yourself with the code



- Exercise one: counting visitors
  - The goal of this exercise is to fix a unit test
    - `zooKeepsCountOfAllVisitors()`
  - The following tasks are needed for this:
    1. Enable the Spring AOP Engine
    2. Create class `BookKeeping` as an aspect and a Spring bean
    3. Add an advice that advices the `Zoo.accept(Visitor)` method
    4. Make sure the advice uses the return value of the advised method to keep track of the number of happy and unhappy visitors
  - Do not change any of the unit tests or `Zoo` classes
    - You should rely solely on aspects to achieve your goal
- Exercise two: launching a marketing campaign
  - The goal of this exercise is to fix a unit test:
    - `zooLaunchesMarketingCampaignWhenNewAnimalArrives()`
  - The following tasks are needed for this:
    1. Create a new aspect: `Marketing`
    2. Add an advice that advices the `Zoo` before new animals are added
    3. Trigger the `launchMarketingCampaign()` method with the right parameters. You should figure out where to retrieve the parameters from!
  - Do not change any of the unit tests or `Zoo` classes
    - You should rely solely on Aspects to achieve your goal
- Exercise three: securing the zoo from escaping animals
  - The goal of this exercise is to fix two unit tests:
    - `zooIsAlertedWhenAnimalsEscape()`
    - `zooDoesNotAllowTigersToEscape()`
  - The following tasks are needed for this:
    1. Create a new aspect: `Security`
    2. Create a new advice that prevents all animals except `Chimp`s to escape
    3. When any animal but `Chimp` tries to escape, prevent it by calling `preventEscapeOf()`

4. When a `Chimp` escapes, sound the alert by throwing an `EscapedAnimalException`

- Do not change any of the unit tests or `Zoo` classes
  - You should rely solely on Aspects to achieve your goal

- Exercise four: adding `Restaurant` facilities
  - The goal of this exercise is to a unit test:
    - `zooAddsRestaurantFeaturesNextToCoreBusiness()`
  - The following tasks are needed for this:
    1. Create an implementation of `Restaurant`: `DefaultRestaurant`
    2. Create a new Aspect: `Facilities`
    3. Introduce the `Restaurant` behavior into the exiting `Zoo`
  - Do not change any of the unit tests or `Zoo` classes
    - You should rely solely on Aspects to achieve your goal
    - Take special note that neither `Zoo` nor `Restaurant` are explicitly combined at compile time!

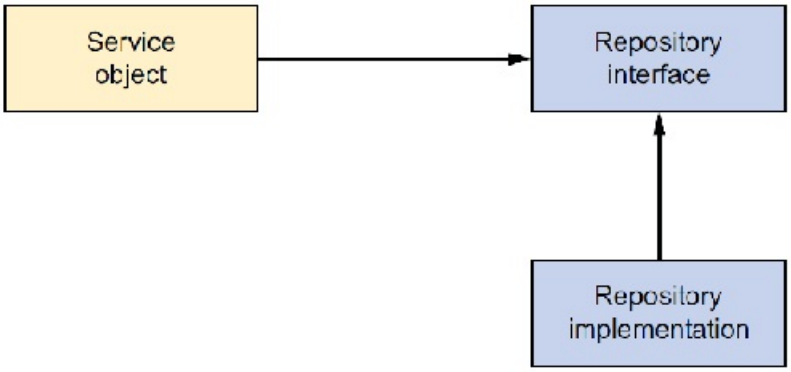
6. Spring Persistence

6.1. Persistence overview

- Persistence is traditionally done using a Relational Database Management System (RDBMS)
- Spring has built-in support for the typical ways to interface with a RDBMS
  - Using low-level JDBC access
  - Using an ORM framework such as JPA
- Alternative persistence mechanisms are also supported by Spring
  - These are known as "NoSQL" or "schema-less" persistence solutions
    - MongoDB, Redis, Neo4j, ...

6.2. Repositories

- Repositories or Data Access Objects (DAO) are specialized objects that deal exclusively with persistence related logic
  - They are used to insulate the rest of the application from persistence layer details
- It is common practice to decouple repository logic from the rest of the application using interfaces



- Decoupling persistence logic through interfaces offers some interesting advantages
  - The application layers become more loosely-coupled
    - Reducing interdependencies, and making the code more reusable
  - It makes the code easier to unit test
    - You can create mocks to "swap out" the real repository with a "fake"
  - The repository interfaces can be made technology agnostic
    - Makes it possible to switch strategies later (for example: switch from JDBC to JPA)
- Spring highly recommends designing to interfaces this way
  - This is by no means enforced though: you can still decide to take a different approach
    - The architect (you!) is still in control of the architecture, not Spring
- Repository interfaces typically have "CRUD"-like methods
  - Create, Read, Update, Delete

```
public interface KnightRepository {
  void create(Knight knight);
  Knight findById(int id);
  List<Knight> findByQuestName(String questName);
  List<Knight> findAll();
  void update(Knight knight);
  void remove(Knight knight);
}
```

- You can create a repository bean using the `@Repository` annotation

```
@Repository
public class JdbcKnightRepository implements KnightRepository {
  // Implement methods here
}
```

6.3. DataAccessException

- Dealing with exceptions in the persistence layer can be challenging
  - There is often no reasonable way to recover from them
    - Unable to connect to database?
    - Invalid query syntax?
    - Constraint violation?
  - They are handled differently by different frameworks
    - JDBC throws a single checked `SQLException`
    - JPA throws an entire range of unchecked `PersistenceException` subclasses
  - They should and should not be handled by the repository layer
    - Not doing so would defeat the purpose of "isolating" the persistence layer
    - But often the services need to at least know that there is a problem
- Spring translates all persistence layer exception types into its own exception hierarchy
- This hierarchy solves the previous challenges
  - They are unchecked
    - Requiring only catch blocks in case you know what to do with them
  - They are framework agnostic
    - Not leaking any framework specific exceptions, thus keeping the persistence layer insulated
  - They are semantically rich
    - Allowing you to differentiate between different types of exceptions (catch only what you want to fish for)
- Spring's exception hierarchy roots from `DataAccessException` and has many subclasses

BadSqlGrammarException	PessimisticLockingFailureException
DataIntegrityViolationException	QueryTimeoutException
DataRetrievalFailureException	RecoverableDataAccessException
DataSourceLookupApiUsageException	SQLWarningException
DeadlockLoserDataAccessException	SqlXmlFeatureNotImplementedException
DuplicateKeyException	**TransientDataAccessException**
EmptyResultDataAccessException	TransientDataAccessResourceException
InvalidDataAccessApiUsageException	TypeMismatchDataAccessException
OptimisticLockingFailureException	UncategorizedDataAccessException
PermissionDeniedDataAccessException	UncategorizedSQLException

- Example without `DataAccessException`

```
public interface KnightRepository {
  void create(Knight knight) throws SQLException; // Persistence layer not insulated!
}
```

```
try { repository.create(knight); } catch(SQLException e) {
  // Must catch! Leaking persistence layer details (JDBC) to the service layer
```

```
}
```

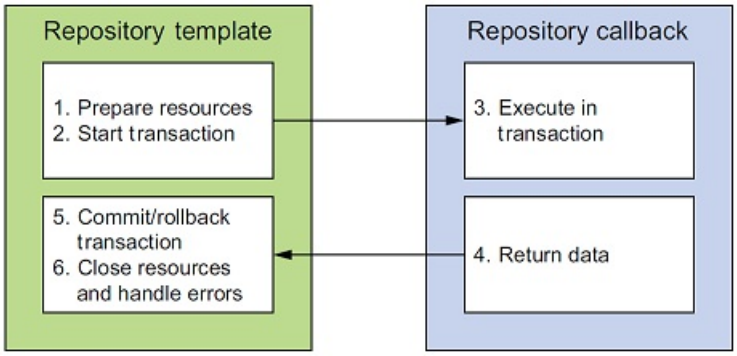
- Example with `DataAccessException`

```
public interface KnightRepository {
  void create(Knight knight); // Insulated, yet still able to catch when needed
}
```

```
try { repository.create(knight); } catch(OptimisticLockingFailureException e) {
  // Handle specific exception, in case you need to!
}
```

6.4. Template classes

- Spring integrates with most persistence technologies using "template" classes, providing many convenient features
  - Reducing boilerplate code
  - Simplifying typical usage scenarios to "one liners"
  - Taking care of exception translation to `DataAccessException`
  - Still allowing access to the full power of the underlying framework using callbacks
- Various template classes exist, each for integration with a specific framework
  - `JdbcTemplate`
  - `JpaTemplate`
  - `HibernateTemplate`
- Spring's templates are based on the "template method" design pattern
  - This pattern allows a bigger problem to be split in a sequence of smaller sub-problems
  - Each smaller problem can optionally be customized (overridden) using a callback



- Note:
  - This is frequently implemented using polymorphism or anonymous inner classes, but since Java 8 you can conveniently use lambdas

6.5. DataSource

- Every application that connects to a RDBMS needs a `DataSource`
  - `DataSource` is an abstraction around the JDBC connection properties needed to connect to the database
    - Required: URI, username, password, driver class
    - Optional: connection pooling parameters, ...
- Through the `DataSource`, applications can obtain database connections

```
public interface DataSource { // Simplified for conceptual idea
  Connection getConnection() throws SQLException;
}
```

- The bottom line is: any Java persistence technology that connects to a RDBMS will need to have a `DataSource`
- In a Spring application, the `DataSource` is configured as a bean in the `ApplicationContext`
  - All we need to do is make sure an implementation of `DataSource` is in the context
- There are several ways you can obtain a `DataSource`
  - Using JDBC's `DriverManager`
  - Using a connection pooled configuration
  - Using a JNDI lookup
  - Using an embedded `DataSource`

6.6. DriverManagerDataSource

- Simple applications can create a `DataSource` by using JDBC's `DriverManager` class underneath
  - Advantage: extremely simple
  - Disadvantage: not considered production grade (no concurrency support)
    - Still useful in unit tests for example

```
@Bean
public DataSource dataSource() {
  DriverManagerDataSource ds = new DriverManagerDataSource();
  ds.setDriverClassName("org.h2.Driver");
  ds.setUrl("jdbc:h2:tcp://localhost/spitter");
  ds.setUsername("sa");
  ds.setPassword("");
  return ds;
}
```

6.7. Pooled DataSource

- Almost all Spring applications benefit from a connection pooled `DataSource`
  - A separate Connection pooling library is required for this: C3P0, Commons DBCP, ...
  - Advantage: almost as simple to configure as a `DriverManagerDataSource`, but without the disadvantages
  - Disadvantage: not externalized, requires rebuilding upon changes

```
@Bean
public BasicDataSource dataSource() {
  BasicDataSource ds = new BasicDataSource(); // From Commons DBCP
  ds.setDriverClassName("org.h2.Driver");
  ds.setUrl("jdbc:h2:tcp://localhost/spitter");
  ds.setUsername("sa");
  ds.setPassword("");
  ds.setInitialSize(5);
  ds.setMaxActive(10);
  return ds;
}
```

6.8. JNDI DataSource

- When a Java EE application server, database connection settings are usually managed by the server
  - The `DataSource` can be obtained by performing a JNDI lookup
  - Advantage: managed by the server, and can be changed without recompiling

- Disadvantage: a Java EE application server is required for this

```
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/jdbc/SpitterDS" />
```

```
@Bean
public JndiObjectFactoryBean dataSource() {
  JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();
  jndiObjectFB.setJndiName("java:comp/jdbc/SpitterDS");
  jndiObjectFB.setProxyInterface(javax.sql.DataSource.class);
  return jndiObjectFB;
}
```

6.9. Embedded DataSource

- Some simple applications ship with their own embedded database
  - A popular example is H2, which can run in "in-memory" or "file-based" mode

```
@Bean
public DataSource dataSource() {
  return new EmbeddedDatabaseBuilder()
    .setType(EmbeddedDatabaseType.H2)
    .addScript("classpath:schema.sql")  // Executed when initialized
    .addScript("classpath:test-data.sql")
    .build();
}
```

- These are often used to run unit tests
  - Extremely fast
  - Simple to reinitialize upon each test run

6.10. Using profiles

- It would be interesting to conditionally select the `DataSource` to use depending on the environment
  - This is where Spring's profiles come in handy

```
@Profile("development")
@Bean
public DataSource embeddedDataSource() {
  return new EmbeddedDatabaseBuilder()
    .setType(EmbeddedDatabaseType.H2).addScript("classpath:test-data.sql").build();
}

@Profile("production")
@Bean
public DataSource dataSource() {
  JndiObjectFactoryBean jndiObjectFactoryBean = new JndiObjectFactoryBean();
  jndiObjectFactoryBean.setJndiName("jdbc/SpittrDS");
  jndiObjectFactoryBean.setResourceRef(true);
  jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
  return (DataSource) jndiObjectFactoryBean.getObject();
}
```

7. Spring JDBC

7.1. JDBC pros and cons

- JDBC is a low-level data access API
  - It offers a way to execute SQL from Java
    - It does not make abstraction of the SQL statements
- Advantages
  - No extra data abstraction frameworks to learn
  - You can access the database's proprietary features
  - Full control and tweaking of performance
- Disadvantages
  - No convenience features
  - Lots of error handling and boilerplate code
- To use `JdbcTemplate` you need to add a dependency to your project

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>4.1.5.RELEASE</version>
</dependency>
```

- You may also need a JDBC Driver for the corresponding database
  - This depends whether you use a Java EE server or not

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.33</version>
</dependency>
```

7.2. JDBC boilerplate code

- Traditional JDBC code requires lots of boilerplate logic
  - The error handling can not be skipped, as this would lead to resource leaks!

```
public void addSpitter(Spitter spitter) {
  Connection conn = null; PreparedStatement s = null;
  try {
    conn = dataSource.getConnection();
    s = conn.prepareStatement("insert into spitter (username,password,fullname) values (?,?,?)");
    s.setString(1, spitter.getUsername());
    s.setString(2, spitter.getPassword());
    s.setString(3, spitter.getFullname());
    s.execute();
  } catch (SQLException e) { // Handle exception
  } finally {
    try {
      if (conn != null) { // Clean up resources
        conn.close();
      }
    } catch (SQLException e) { // Handle critical exception
    }
  }
}
```

- Another example to prove the point
  - Error handling can not be skipped, but is nearly identical to the previous example

```
public Spitter findOne(long id) {
```

```
  Connection conn = null; PreparedStatement s = null; ResultSet rs = null;
  try {
    conn = dataSource.getConnection();
    s = conn.prepareStatement("select id, username, fullname from spitter where id=?");
    s.setLong(1, id)
    rs = s.executeQuery();
    if (rs.next()) {
      Spitter spitter= new Spitter();
      spitter.setId(rs.getLong("id"));
      spitter.setUsername(rs.getString("username"));
      spitter.setFullname(rs.getString("fullname"));
      return spitter;
    }
  } catch (SQLException e) { /* Handle exception */ }
  finally {
    try { if(conn != null) conn.close(); } catch(SQLException e) { /* Handle critical exception */ }
  }
  return null;
}
```

7.3. JdbcTemplate

- Spring provides the `JdbcTemplate` class to facilitate working with JDBC
  - Uses the "template method" design pattern
- `JdbcTemplate` cleans up the nasty parts of JDBC, leaving only the advantages
  - Abstracts away boilerplate logic
    - Exception handling, closing connections, ...
  - Provides easy to reuse convenience features
- This way applications can focus more on business value, and less on implementation technicalities

7.4. Configuring JdbcTemplate

- Using `JdbcTemplate` is a matter of injecting a reference to a `JdbcTemplate` bean into your repository
  - `JdbcTemplate` itself depends on a `DataSource`

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new JdbcTemplate(dataSource);
}

@Bean
public DataSource dataSource() {
  return new DriverManagerDataSource("jdbc:mysql:/localhost/spitter", "user", "pass");
}
```

```
@Repository
public class JdbcSpitterRepository implements SpitterRepository {
  @Autowired JdbcTemplate jdbcTemplate;
}
```

7.5. Eliminating boilerplate code

- The two previous examples can now be rewritten using `JdbcTemplate`
  - Spring handles and converts `SQLException` to `DataAccessException` automatically

```
public void addSpitter(Spitter spitter) {
```

```
  jdbcTemplate.update("insert into spitter(username, password, fullname) values(?, ?, ?)",
  spitter.getUsername(),
  spitter.getPassword(),
  spitter.getFullname());
}
```

```
public Spitter findOne(long id) {
  return jdbcTemplate.queryForObject(
  "select id, username, fullname from spitter where id=?", (rs, rowNum) -> {
    return new Spitter(
    rs.getLong("id"),
    rs.getString("username"),
    rs.getString("fullname"));
  }, id);
}
```

7.6. RowMappers

- `RowMapper`s are used by `JdbcTemplate` to convert a record to an object
  - They describe how `JdbcTemplate` should convert a JDBC `ResultSet` at a predefined position (record) into an arbitrary object
  - They can be reused if defined as a named class or variable

```
jdbcTemplate.queryForObject("select * from spitter where id = ?", new RowMapper<Spitter>() {
  public Spitter mapRow(ResultSet rs, int rowNum) throws SQLException {
    Spitter spitter = new Spitter();
    spitter.setId(rs.getInt("id"));
    spitter.setUsername(rs.getString("username"));
    spitter.setPassword(rs.getString("password"));
    spitter.setEmail(rs.getString("email"));
    spitter.setFullname(rs.getString("fullname"));
    return spitter;
  }
}, id);
```

- When using Java 8, you can reduce amount of the code by using lambdas

```
RowMapper<Spitter> mapper = (rs, rowNum) -> {
  Spitter spitter = new Spitter();
  spitter.setId(rs.getInt("id"));
  spitter.setUsername(rs.getString("username"));
  spitter.setPassword(rs.getString("password"));
  spitter.setEmail(rs.getString("email"));
  spitter.setFullname(rs.getString("fullname"));
  return spitter;
}
jdbcTemplate.queryForObject("select * from spitter where id=1", mapper);
```

- Or even shorter when using a method reference to an existing method

```
jdbcTemplate.queryForObject("select * from spitter where id=1", SpitterRepository::map);
```

```
static Spitter map(ResultSet rs, int rowNum){ ... }
```

7.7. JdbcTemplate API

- `JdbcTemplate` has many convenient methods for various SQL statements
  - Most have a number of overloaded variants that allow extra arguments to be passed

Method	Description

`execute()`	For generic queries that do not return anything, such as `CREATE`, `ALTER`,...
`query()`	For queries that return lists of composites such as `SELECT * FROM spitter`
`queryForList()`	For queries that return lists of simple types such as `SELECT birth_date FROM spitter`
`queryForObject()`	For queries that return a single record such as `SELECT * FROM spitter WHERE id=1`
`queryForMap()`	For queries that return "flat" records for which there is no convenient class to map to
`batchUpdate()`	For efficient execution of batch statements
`update()`	For `INSERT`, `DELETE` and `UPDATE` statements (that return an "affected row count")

7.8. Named parameters

- JDBC only supports indexed parameters natively

```
INSERT INTO spitter(username, password, fullname) VALUES(?, ?, ?)
```

- Spring extends this by adding named parameter support
  - Very convenient when a query requires a lot of parameters to be filled

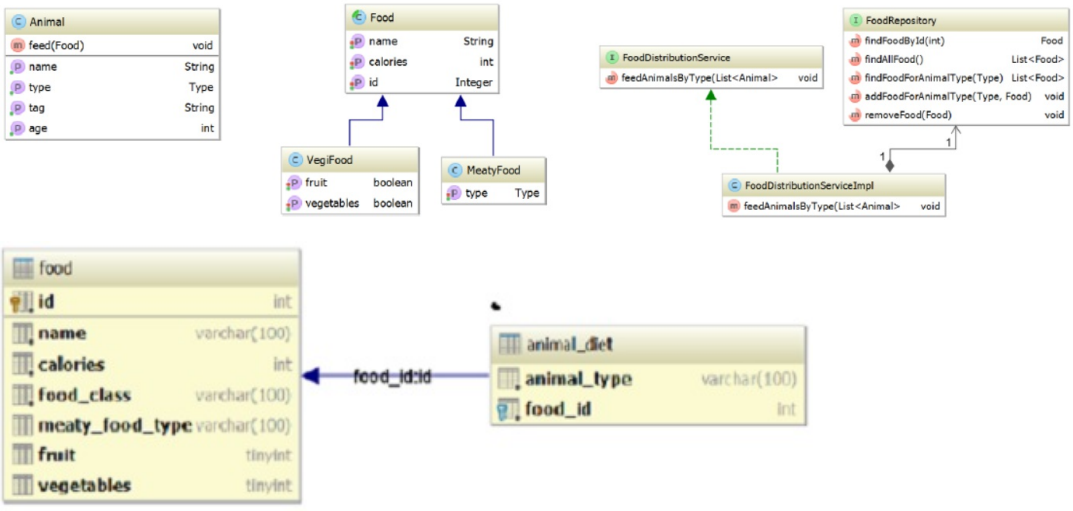
```
INSERT INTO spitter(username, password, fullname) VALUES(:username, :password, :fullname)
```

- To use this, you should inject a `NamedParameterJdbcTemplate`

```
Map<String, Object> paramMap = new HashMap<String, Object>();
paramMap.put("username", spitter.getUsername());
paramMap.put("password", spitter.getPassword());
paramMap.put("fullname", spitter.getFullname());
namedParameterJdbcTemplate.update(SQL, paramMap);
```

7.9. Exercise: Spring JDBC

- Open the "data-access-jdbc" exercise provided by the teacher
  - Explore the project to familiarize yourself with the code and the database structure



- Exercise one: configuring a `DataSource` and `JdbcTemplate`
  - We will need two `DataSource` configurations, one for "test" and one for "production"
    1. Add an Apache DBCP `Datasource` to the "production" profile
    2. Add an embedded H2 `Datasource` to the "test" profile

    - Next, add a `JdbcTemplate` that used (one of) the previously configured `DataSource`s

- Exercise two: implement the `FoodRepository` for JDBC
    - The goal of this exercise is to fix the unit tests and the `Main` application
    - The following steps are needed for this:
        1. Create a repository bean that implements the provided `FoodRepository`
        2. Inject a `JdbcTemplate` into your `FoodRepository`
        3. Implement `findAllFoods()`, and create a `RowMapper` to help you
        4. Implement `findFoodById()`, and reuse your `RowMapper`
        5. Implement `findFoodForAnimalType()`, and reuse your `RowMapper`
        6. Implement `removeFood()`
    - The unit tests will guide your way
        - Do not modify them!
    - When done, run the `Main` application and see what happens!

8. Spring JPA

8.1. JPA pros and cons

- JPA extends on JDBC by adding an extra layer of abstraction on top of it
  - This abstraction is called Object Relational Mapping (ORM)
    - Other ORM frameworks are: Hibernate, iBATIS, ...
- Advantages
  - Can simplify tedious work with JDBC
    - Lazy and eager loading
    - Cascading rules and transparent joins
  - Offers an object perspective to relational data
- Disadvantages
  - An extra layer of abstraction to learn
  - Less control over tweaking and performance

8.2. Spring ORM

- Spring ORM supports multiple ORM frameworks
  - We will use the official standard Java Persistence API (JPA)
    - Starting from Spring 4, a JPA 2 runtime library is required
- The ORM module supports features such as
  - Transactions
  - Translation of framework exceptions to `DataAccessException`
  - Template classes
  - DAO Support classes
- In order to use JPA, you will need to add some dependencies

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-orm</artifactId>
  <version>4.x.x.RELEASE</version> <!-- Select latest version -->
</dependency>
```

- You also need a JPA implementation and a JDBC driver

```
<dependency>
    <groupId>org.hibernate</groupId> <!-- Hibernate is a JPA implementation -->
    <artifactId>hibernate-entitymanager</artifactId>
    <version>x.x.x.Final</version> <!-- Select latest version -->
</dependency>
<dependency>
    <groupId>mysql</groupId> <!-- Replace with your own database driver! -->
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.xx</version>
</dependency>
```

8.3. POJO repositories

- JPA provides the `EntityManager` class as the gateway to the persistence world
  - It is like a "JDBC connection on steroids"; most database interactions go through this interface
- JPA has much less boilerplate logic to deal with, so there is no need for a template class like `JdbcTemplate`
- Instead, Spring JPA will use a pure POJO approach to inject an `EntityManager` into a repository

- This makes a Spring repository identical to an EJB or CDI bean
  - The only difference is some annotation metadata

8.4. EntityManagerFactory

- All JPA applications need to have an `EntityManagerFactory` configured
  - Responsible for creating `EntityManager` instances injected into the repositories
- There are two types of `EntityManagerFactory`
  - Application managed (`RESOURCE_LOCAL`)
    - Used outside a Java EE application server
  - Container managed (`JTA`)
    - Used on a Java EE application server like WildFly or WebLogic
- There are two ways to configure each of them using Spring
  - Using JPA's own `/META-INF/persistence.xml`
  - Using Spring's Java configuration

8.5. Spring based persistence configuration

- Spring 4 provides a no-xml configuration style for the `EntityManagerFactory`
  - This way you do not need a `persistence.xml` at all

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean(DataSource dataSource,
  JpaVendorAdapter jpaVendorAdapter) {
  LocalContainerEntityManagerFactoryBean emfb = new LocalContainerEntityManagerFactoryBean();
  emfb.setDataSource(dataSource);
  emfb.setJpaVendorAdapter(jpaVendorAdapter);
  emfb.setPackagesToScan("com.realdolmen");
  return emfb;
}
```

- Notes:
  - You can specify the `DataSource` bean to use (including one obtained using JNDI)
  - You can specify which packages to scan for JPA entities
  - You can specify JPA implementation configuration specifics using a `JpaVendorAdapter`
- The `JpaVendorAdapter` allows configuration of vendor properties
  - Several are available to choose from:
    - `HibernateJpaVendorAdapter`, `EclipseLinkJpaVendorAdapter`, `OpenJpaVendorAdapter`
  - The database dialect can be selected with the `database` property
    - `DEFAULT`, `DB2`, `DERBY`, `H2`, `HSQL`, `INFORMIX`, `MYSQL`, `ORACLE`, `POSTGRESQL`, `SQL_SERVER`, `SYBASE`

```
@Bean
public JpaVendorAdapter jpaVendorAdapter() {
  HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
  adapter.setDatabase(Database.MYSQL);
  adapter.setGenerateDdl(true);
  adapter.setShowSql(true);
  return adapter;
}
```

8.6. JPA based persistence configuration

- Spring can still be configured using a classic JPA `persistence.xml` configuration
- With the persistence configuration fixed in `/META-INF/persistence.xml`

```
@Bean
public LocalEntityManagerFactoryBean entityManagerFactoryBean() {
  return new LocalEntityManagerFactoryBean();
}
```

- With the persistence configuration in an arbitrary location

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean() {
  LocalContainerEntityManagerFactoryBean emfb = new LocalContainerEntityManagerFactoryBean();
  emfb.setPersistenceXmlLocation("/my/strange/location/for/persistence.xml");
  return emfb;
}
```

The `persistence.xml` not shown here is specific for JPA.

8.7. Transaction manager

- JPA does not have transactions set to "auto commit" by default
  - This means that in order to modify any data, we need to setup a Spring transaction manager, and mark repositories as `@Transactional`

```
@Bean
public JpaTransactionManager transactionManager (EntityManagerFactory entityManagerFactory) {
  return new JpaTransactionManager(entityManagerFactory);
}
```

- You also need to enable transactions with `@EnableTransactionManagement`

```
@Configuration
@EnableTransactionManagement
public class PersistenceConfig { /* ... */ }
```

8.8. JPA repositories

- When the `EntityManagerFactory` is configured, we can inject `EntityManager` instances into a repository using JPA's `@PersistenceContext` annotation

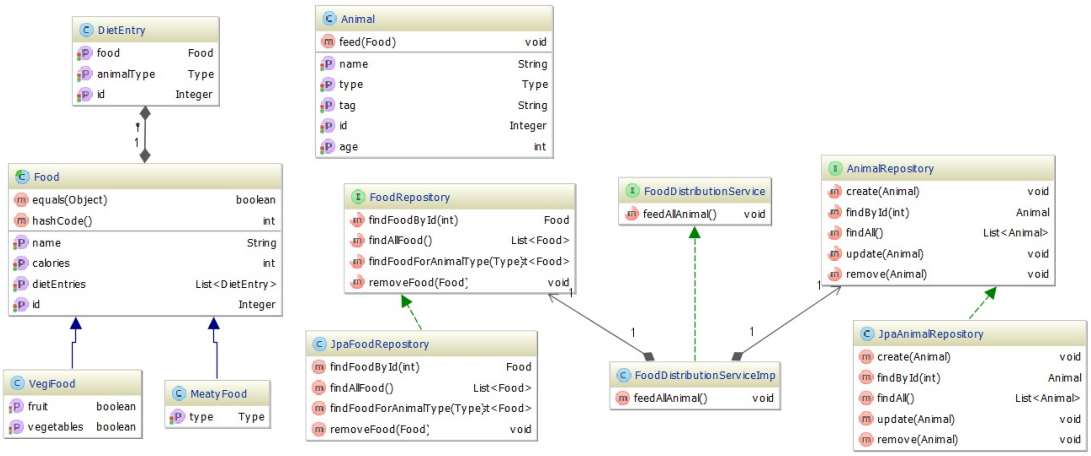
```
@Repository
@Transactional
public class JpaSpitterRepository implements SpitterRepository {
  @PersistenceContext private EntityManager em;

  public void addSpitter(Spitter spitter) {
    em.persist(spitter);
  }

  public Spitter getSpitterById(Long id) {
    return em.find(Spitter.class, id);
  }

  public void saveSpitter(Spitter spitter) {
    em.merge(spitter);
  }
}
```

- The `Spitter` class in the previous example must be a valid JPA entity

```java
@Entity
public class Spitter {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String username;

  private String fullname;

  private String email;

  private String password;

  // Add constructors, getters and setters as well
}
```

8.9. Exercise: Spring JPA

- Open the "data-access-jpa" exercise provided by the teacher



- Exercise one: configuring an `EntityManagerFactory`
  - The goal of this exercise is to configure Spring to support the use of JPA
  - The following steps are needed for this:
    1. Configure a DBCP based MySQL `DataSource` for the production profile
    2. Configure an embedded H2 `DataSource` for the test profile
    3. Configure an `EntityManagerFactoryBean` for Hibernate, using only Java configuration
    4. Make sure the production profile uses the `MYSQL` database dialect
    5. Make sure the test profile uses the `H2` database dialect
- Exercise two: implementing the `JpaAnimalRepository`
  - A domain class `Animal` already exists, but is not usable with JPA yet
  - The goal of this exercise is to fix all the unit tests in `AnimalRepositoryTest`
    - Do not modify the unit tests!
  - To do this, the following steps are required:
    1. Enhance the `Animal` class so it can be used as a JPA entity
    2. Implement `AnimalRepository` as a transactional repository that injects a persistence context
    3. Implement `AnimalRepository` contract using JPA
- Exercise three: implementing the `JpaFoodRepository`
  - The domain classes for `Food`, `VegiFood`, `MeatyFood` and `DietEntry` have already been configured for JPA, take a look at the annotations used!
  - The goal of this exercise is to fix all the unit tests for `FoodRepositoryTest`
    - Do not modify any of the unit tests!

- To do this, the following steps are needed:
  1. Implement `FoodRepository` as a new transactional repository that injects a persistence context
  2. Implement all the CRUD methods defined by the interface using JPA
- Note: this exercise requires some more knowledge about JPA

9. Spring Data

9.1. Spring Data Motivation

- Creating repositories can become repetitive
  - The method below will probably look the same for any entity
  - The domain type may be different, but the method is fairly common across repositories

```
public void addSpitter(Spitter spitter) {
  entityManager.persist(spitter);
}
```

- Spring Data JPA brings an end to boilerplate repository methods
  - Rather than write the same repository implementations again and again, you can stop at writing the repository interface
  - You then get access to 18 methods for performing common persistence operations

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

9.2. Spring Data Configuration

- The repository class itself will be created by Spring Data at startup time

```
@Configuration
@EnableJpaRepositories(basePackages="com.acme.spittr.db")
public class JpaConfiguration {
  //...
}
```

- The implementation includes the 18 methods from
  - `JpaRepository`
  - `PagingAndSortingRepository`
  - `CrudRepository`
- But what if you need more than these 18 methods?

9.3. Defining query methods

- You only need to add a method to the interface

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

- Spring Data will examine the method and attempt to understand its purpose
  - Spring Data defines its own DSL where persistence details are expressed in the method signatures
- Repository methods are composed of a verb, and optional subject, the word **`By`**, and a predicate
  - E.g. `readSpitterByFirstnameOrLastname()`
  - `get`, `read` and `find` are synonyms
  - There is also a `count` verb to retrieve a count of matching objects
  - When the subject begins with `Distinct`, distinct results will be returned
- With the predicate, you can constrain the results
  - Each condition references a property and may specify a comparison operation
  - If the comparison operator is left out, it implied to be an equals operations

- Other predicates
  - `IsAfter`, `After`, `IsBetween`, `Between`
  - `IsGreaterThan`, `GreaterThan`, `IsGreaterThanEqual`, `GreaterThanEqual`, `IsBefore`, `Before`, `IsLessThan`, `LessThan`, `IsLessThanEqual`, `LessThanEqual`
  - `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`
- The values that the properties will be compared against are the parameters of the method

```
List<Spitter> readByFirstnameOrLastname(String first, String last);
```

- Case-insensitive search are done with `IgnoringCase` or `IgnoresCase`

```
List<Spitter> readByFirstnameIgnoringCaseOrLastnameIgnoresCase(String first, String last);
```

```
List<Spitter> readByFirstnameOrLastnameAllIgnoresCase(String first, String last);
```

- Sorting is done using `OrderBy` and `Asc` or `Desc`

```
List<Spitter> readByFirstnameOrLastnameOrderByLastnameAscFirstnameDesc(String first, String last);
```

- By carefully constructing a repository method signature, you can let Spring Data JPA generate an implementation to query almost anything!

9.4. Declaring custom queries

- Imagine the following method
  - This searches for all `Spitter`s whose email address is a Gmail address
  - This method does not adhere to the Spring Data method naming conventions

```
List<Spitter> findAllGmailSpitters();
```

- In these situations, you can give Spring Data the query that should be performed
  - This is also useful in situations when the method name would become too long

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

```
// Should replace this method with a @Query
List<Order> findByCustomerAddressZipCodeOrCustomerNameAndCustomerAddressState();
```

9.5. Mixing in custom functionality

- When you need any custom functionality that Spring Data does not offer…
  - … just use the `EntityManager` at lower level on the methods that need it
  - You can still keep using Spring Data for the things it knows how to do
- Create a class with the same name as the interface, suffixed with `Impl`
  - If the class exists, Spring Data JPA merges its methods with the generated methods

```
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 Spitters WHERE (SELECT COUNT(spittles) FROM s.spittles spittles) >" + "10000)";
    return em.createQuery(update).executeUpdate();
  }
}
```

- Notice the `SpitterRepositoryImpl` does not implement `SpitterRepository`
  - Spring Data JPA is still responsible for that interface
  - The only thing that ties your implementation to Spring Data is its name
- Make sure that the interface you implement is declared in `SpitterRepository`
  - The easiest way is by making the `SpitterRepository` interface extend it

```
public interface SpitterSweeper {
  int eliteSweep();
}
```

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

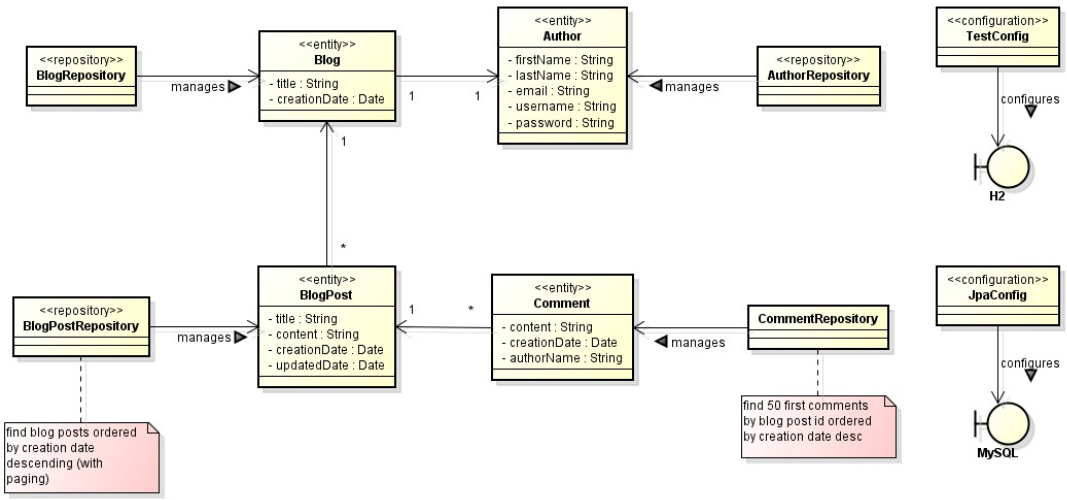
- If you prefer using another suffix than `Impl`, you can configure it

```
@EnableJpaRepositories(
  basePackages="com.acme.spittr.db",
  repositoryImplementationPostfix="Helper"
)
```

- Spring Data will now search for a class named `SpitterRepositoryHelper`
- Spring Data JPA is just the tip of the iceberg!
  - It also supports NoSQL databases!

9.6. Exercise: Spring Data

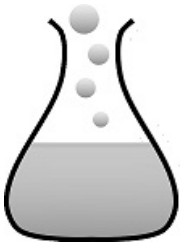
- Create a domain for a Blog with following JPA entity classes:
  - `Blog`
  - `Author`
  - `BlogPost`
  - `Comment`
- Create repositories using Spring Data
- Add methods to
  - Find all blog posts ordered by creation date descending (with paging)
  - Find the first 50 comments by blog post id ordered by creation date descending
- Create a "test" and "production" profile, one with embedded database, the other for MySQL
- Test the repositories with JUnit tests


<<repository>>
BlogRepository
manages
<<entity>>
Blog
- title : String
- creationDate : Date
1
1
<<entity>>
Author
- firstName : String
- lastName : String
- email : String
- username : String
- password : String
manages
<<repository>>
AuthorRepository
<<configuration>>
TestConfig
configures
H2
1
*
<<repository>>
BlogPostRepository
manages
<<entity>>
BlogPost
- title : String
- content : String
- creationDate : Date
- updatedDate : Date
1
*
<<entity>>
Comment
- content : String
- creationDate : Date
- authorName : String
manages
CommentRepository
<<configuration>>
JpaConfig
configures
MySQL
find blog posts ordered by creation date descending (with paging)
find 50 first comments by blog post id ordered by creation date desc

10. Transaction Support

10.1. Understanding transactions

- A transaction is a unit of work performed against a database
- Transactions generally represent changes in the database
  - They provide recovery from failure and keep databases consistent
  - They provide isolation between programs accessing the database concurrently
- Transactions are ACID
  - A tomic: all or nothing
  - C onsistent: leave the database in a valid state
  - I solated: determines the visibility to other transactions
  - D urable: once committed, the results are stored permanently



- An example transaction: ATM money withdrawal
  - Verify account details
  - Accept withdrawal request
  - Check balance
  - Update balance
  - Dispense money
- If any of these steps fail, we should rollback any changes made
- If all steps succeed, we can commit the transaction

10.2. Transaction support

- Transaction support is one of the main reasons to use Spring
  - Consistent across different APIs (JDBC, Hibernate, JPA, JTA, ...)
  - Support for declarative transaction management
  - Simpler API for programmatic transaction management
  - Easy to integrate with Spring data access abstractions
- Spring resolves the disadvantages of local and global transactions
  - Local: resource-specific, invasive, cannot work across multiple resources
  - Global: only work with JTA and JNDI, cumbersome (`Exception`s), need an application server
- With Spring, you write code once, and you can benefit from different management strategies in different environments

10.3. Spring framework transaction strategy

- The transaction strategy is defined by the `PlatformTransactionManager`

```
public interface PlatformTransactionManager {
  TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;
  void commit(TransactionStatus status) throws TransactionException;
  void rollback(TransactionStatus status) throws TransactionException;
}
```

- As an interface, it can be easily mocked or stubbed
  - It is not tied to a lookup strategy such as JNDI
  - Transactional code can be tested much more easily than if using JTA directly
- Spring's implementations are defined like any other bean
- `TransactionException` is unchecked, so you are not forced to catch it

10.4. Choosing a platform

- You need to define a `PlatformTransactionManager` implementation
  - This implementation requires knowledge of the environment in which it will work
- Implementations
  - JDBC: `DataSourceTransactionManager`
  - JTA: `JtaTransactionManager`
  - Hibernate: `HibernateTransactionManager`
  - JPA: `JpaTransactionManager`
- Whatever you choose, configuration is similar
  - In all cases, application code does not need to change

10.5. Defining transaction managers

- Add the bean to your configuration depending on the chosen persistence
- JDBC

```
@Bean
public PlatformTransactionManager transactionManager(DataSource dataSource) {
  return new DataSourceTransactionManager(dataSource);
}
```

- JTA
  - This bean will look the same regardless the chosen technology, due to the fact that JTA transactions are global transactions

```
@Bean
public PlatformTransactionManager transactionManager() {
  return new JtaTransactionManager();
}
```

- Hibernate

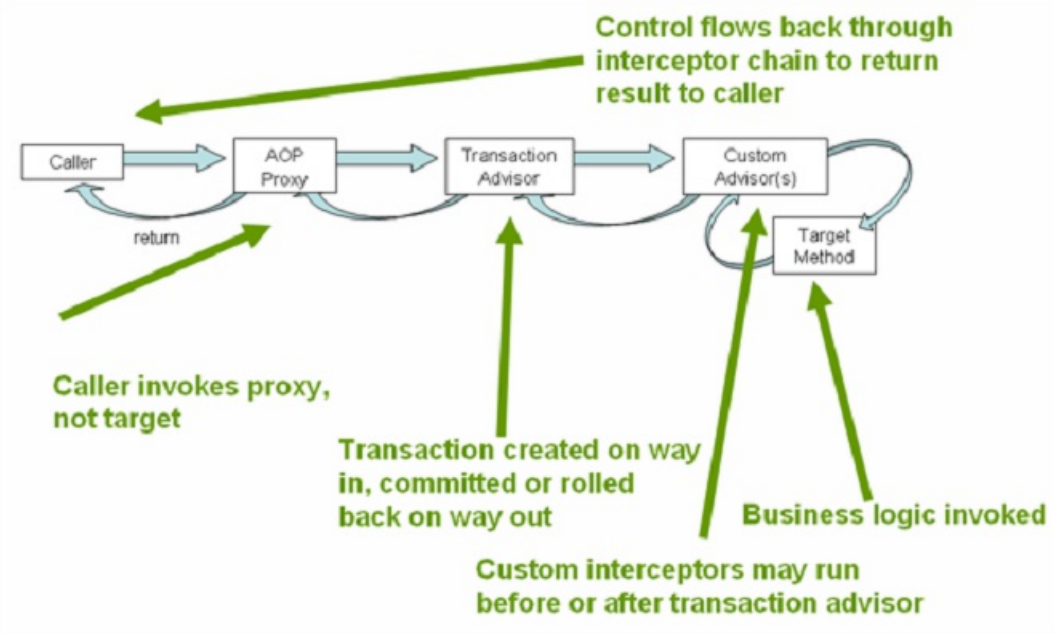
```
@Bean
public PlatformTransactionManager transactionManager(SessionFactory sessionFactory) {
  return new HibernateTransactionManager(sessionFactory);
}
```

- JPA

```
@Bean
public JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {
  return new JpaTransactionManager(entityManagerFactory);
}
```

10.6. Declarative transaction management

- This is used by most Spring users
  - This option has the least impact on application code and is the most consistent with the ideals of a non-invasive lightweight container
- Declarative transactions are made possible with Spring AOP
- You can even declaratively specify for which `Exception`s to rollback
  - This is part of the configuration
  - The advantage is that business objects do not depend on the transaction infrastructure



10.7. Using @Transactional

- Consider the following interface and implementation
  - With the `@Transactional` annotation, this bean can be made transactional

```
public interface FooService {
  Foo getFoo(String fooName);
  void insertFoo(Foo foo);
  void updateFoo(Foo foo);
}
```

```
@Service
@Transactional
public class DefaultFooService implements FooService {
  public Foo getFoo(String fooName) {
    throw new UnsupportedOperationException();
  }
  public void insertFoo(Foo foo) {
    throw new UnsupportedOperationException();
```

```
  }
  public void updateFoo(Foo foo) {
    throw new UnsupportedOperationException();
  }
}
```

- You still need to enable transactions in JavaConfig, or they won't happen

```
@Configuration
@EnableTransactionManagement
public class PersistenceConfig{
  @Bean
  public JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory){
    return new JpaTransactionManager(entityManagerFactory);
  }
  // ...
}
```

- `@Transactional` can be placed
  - before an interface definition
  - on a method on an interface
  - on a class definition (preferred)
  - on a public method on a class
- The most derived location takes precedence when evaluating the transactional settings for a method

```
@Service
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

  public Foo getFoo(String fooName) {
    // do something
  }

  @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
  public void updateFoo(Foo foo) {
    // do something
  }
}
```

10.8. @Transactional settings

- Defaults
  - Propagation: `PROPAGATION_REQUIRED`
  - Isolation: `ISOLATION_DEFAULT`
  - Transaction is read/write
  - Default timeout of the underlying transaction system is used
  - `RuntimeException`s trigger rollback, checked `Exception`s do not
- All these defaults can be changed with the properties of `@Transactional`

10.9. @Transactional properties

Property	Description
`value`	Optional qualifier specifying the transaction manager to be used
`propagation`	Optional propagation setting
`isolation`	Optional isolation level
`readOnly`	Read/write vs. read-only transaction

`timeout`	Transaction timeout
`rollbackFor`	Optional array of exception classes that must cause rollback
`rollbackForClassName`	Optional array of names of exception classes that must cause rollback
`noRollbackFor`	Optional array of exception classes that must not cause rollback
`noRollbackForClassName`	Optional array of names of exception classes that must not cause rollback

10.10. Transaction propagation

Propagation	Description
`PROPAGATION_MANDATORY`	Method must run within transaction. If no transaction in progress, throw exception.
`PROPAGATION_NESTED`	Method must run in nested transaction. Not all resource manager support this!
`PROPAGATION_NEVER`	Method should never run in a transaction. If transaction in progress, throw exception.
`PROPAGATION_NOT_SUPPORTED`	Method should not run in a transaction. If a transaction in progress, transaction is suspended.
`PROPAGATION_REQUIRED`	Method must run in a transaction. If transaction in progress, run within that transaction; otherwise start new transaction (default).
`PROPAGATION_REQUIRES_NEW`	Method must run in its own transaction. If transaction in progress, it will be suspended for the duration of the method.
`PROPAGATION_SUPPORTS`	Method does not require a transactional context but can run in transaction if already in progress.

10.11. Transaction isolation

Isolation Level	Description
`ISOLATION_DEFAULT`	Use default isolation level of the underlying database.
`ISOLATION_READ_UNCOMMITTED`	Allows to read changes that have not yet been committed. May result in dirty reads, non-repeatable reads and phantom reads.
`ISOLATION_READ_COMMITTED`	Allows to read changes that have been committed. May result in non-repeatable reads and phantom reads.
`ISOLATION_REPEATABLE_READ`	Multiple reads of the same fields will yield the same results unless changed by the transaction itself. Phantom reads might still occur.
`ISOLATION_SERIALIZABLE`	Fully ACID-compliant isolation level. Slowest of all isolation levels because it is typically doing full table locks on tables used in the transaction.

10.12. Programmatic transaction management

- The Spring Framework provides two means of programmatic transaction management
  - Using the `TransactionTemplate` (recommended)
  - Using a `PlatformTransactionManager` implementation directly
- The `TransactionTemplate` works in the same way as other templates
  - It uses a callback approach
  - It will couple your code into Spring's transaction infrastructure
- Using a `TransactionTemplate` will be similar to this

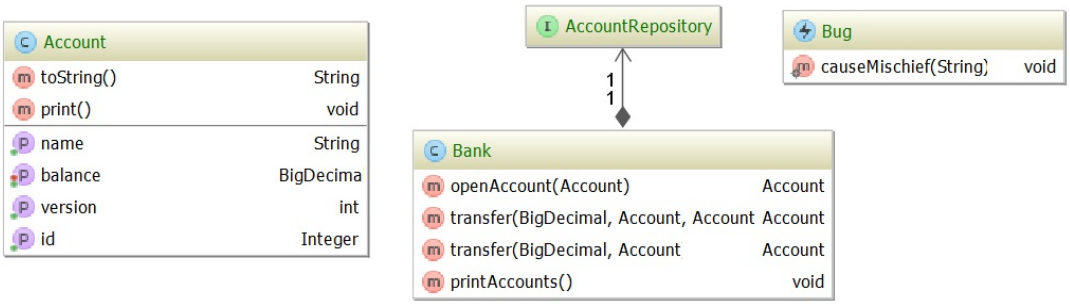
```
@Service
public class SimpleService implements Service {
  private final TransactionTemplate transactionTemplate;

  @Autowired
  public SimpleService(PlatformTransactionManager transactionManager) {
    this.transactionTemplate = new TransactionTemplate(transactionManager);
  }

  public Object someServiceMethod() {
    // using a TransactionCallback as functionalinterface
    return transactionTemplate.execute(ts -> {
      Object result = null;
      // do stuff in a transaction
      return result;
    });
  }
}
```

- From within the callback, you can rollback the transaction

```
return transactionTemplate.execute(ts -> {
  try {
    // do stuff in a transaction
  } catch(SomeBusinessException e) {
    ts.setRollbackOnly();
  }
});
```

10.13. Exercise: Transaction Support

- Open exercise "transactions" provided by the teacher
  - In this exercise we will program a `Bank` that performs a number of transactions on `Account`s
  - Unfortunately there may be some `Bug`s into the money transfer system
  - We need to make sure that even when there are bugs, the `Account` balances are always correct



- Exercise one: configuring a `PlatformTransactionManager`
  - The goal of this exercise is to configure transaction management
  - The following steps are required for this:
    1. Add a `PlatformTransactionManager` bean for JPA
    2. Make sure transaction management is enabled in the Spring context
  - Do you think we are safe now?
    - Continue to exercise two to check!
- Exercise two: making the `Bank` transactional
  1. Explore and run the Main class
     - It will execute some financial transactions for our Bank
  2. Check if the Bank seems to work correctly
     - Do the account balances printed on screen add up?

3. Now let’s do some work on the Bank system, by calling the "maintenance()" method
    - Let’s hope we did not break anything!
4. Check if the Bank still works properly
    - Do the account balances printed on screen still add up?
5. Make the Bank resilient to errors by making it transactional
6. Check again if the numbers are correct!

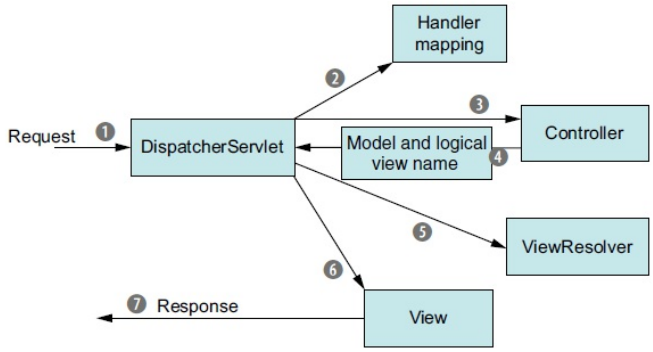
11. Spring Web MVC

11.1. Spring Web MVC

- For many Java developers, web-based applications are their primary focus
- Web applications quickly grow in complexity
- Challenges
  - State management
  - Workflow
  - Validation
  - HTTP is stateless
- Spring Web MVC helps to build web-based applications that are as flexible and loosely coupled as the Spring framework itself

11.2. Getting started with Spring Web MVC

- The Request goes through 7 steps
  1. Front controller `DispatcherServlet`
  2. `HandlerMapping` selects the controller based on the URL of the `Request`
  3. The controller processes the `Request` by delegating to services
  4. Model data and view name is returned
  5. The `DispatcherServlet` consults a `ViewResolver` to find the `View`
  6. The `View` uses the model data to render the output
  7. The `Response` is carried back to the client



11.3. Configuring the DispatcherServlet

- The easiest way is to use Java to configure the `DispatcherServlet`

```
public class SpittrWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
  @Override
  protected String[] getServletMappings() {
    return new String[] { "/" }; // map the DispatcherServlet to '/'
  }
  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] { RootConfig.class };
  }
  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] { WebConfig.class }; // specify a configuration class
  }
}
```

- This requires the Servlet 3.0 specification
  - You do not have an excuse to not have it since it was released in 2009, and is available in Tomcat 7 or higher

11.4. A tale of two application contexts...

- A Spring web application usually has two application contexts

  - The `WebConfig` will define beans for the `DispatcherServlet` (`Controller`s, `ViewResolver`s, `HandlerMapping`s)
  - The `RootConfig` will configure the other beans of the application (middle-and data-tier components that drive the backend)

- The `DispatcherServlet` loads its own configuration
- The backend configuration is loaded by a `ContextLoaderListener`
- The simplest configuration looks as follows

```
@Configuration
@EnableWebMvc
public class WebConfig {
}
```

11.5. Configuring Spring Web MVC

- The previous example is too simple
  - There is no view resolver, component scanning is not enabled and the `DispatcherServlet` will also need to handle static resources ...

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

  @Override
  public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
    configurer.enable(); // configures static resources
  }
}
```

11.6. Configuring the backend application context

- Configure the backend in `RootConfig` as usual
  - Notice we filter out the `WebConfig` configuration by filtering on the `@EnableWebMvc` annotation

```
@Configuration
@ComponentScan(
  basePackages={"spitter"},
  excludeFilters={@Filter(type=FilterType.ANNOTATION, value=EnableWebMvc.class)}
)
public class RootConfig {
}
```

11.7. Writing a simple controller

- Use `@Controller` as stereotype for component scanning
  - Using `@Component` would have the same effect, but it would not be so expressive...

```
@Controller // declare as a controller
public class HomeController {
```

```
  @RequestMapping(value="/", method=RequestMethod.GET) // handle GET requests for '/'
  public String home() {
    return "home"; // the view name is "home"
  }
}
```

- `@RequestMapping` specifies the request path and HTTP method to handle
- The return value will be interpreted as the name of the view to be rendered
  - With the previous `InternalResourceViewResolver`, this is `/WEB-INF/views/home.jsp`

In Spring 4.3 (and with Spring Boot 1.4), new types of `@RequestMapping` annotations have been added to map HTTP methods. So instead of the following example...

```
@RestController
@RequestMapping("/api/books")
public class BookAPIController {
  @RequestMapping
  public ResponseEntity<?> getBooks(){
  }
  @RequestMapping("/{book_id}")
  public ResponseEntity<?> getBook(@PathVariable("book_id") String bookId){
  }
  @RequestMapping(method = RequestMethod.POST)
  public ResponseEntity<?> addNewBook(@RequestBody Map<String, Object> requestBody){
  }
  @RequestMapping(method = RequestMethod.POST, value="/{book_id}")
  public ResponseEntity<?> editBook(@PathVariable("book_id") String bookId){
  }
  @RequestMapping(method = RequestMethod.DELETE, value="/{book_id}")
  public ResponseEntity<?> deleteBook(@PathVariable("book_id") String bookId){
  }
}
```

...we can now use the alternative annotations `@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping` and `@DeleteMapping`. This makes the below code a lot easier to read:

```
@RestController
@RequestMapping("/api/books")
public class BookAPIController {
  @GetMapping
  public ResponseEntity<?> getBooks(){
  }
  @GetMapping("/{book_id}")
  public ResponseEntity<?> getBook(@PathVariable("book_id") String bookId){
  }
  @PostMapping
  public ResponseEntity<?> addNewBook(@RequestBody Map<String, Object> requestBody){
  }
  @PostMapping("/{book_id}")
  public ResponseEntity<?> editBook(@PathVariable("book_id") String bookId){
  }
  @DeleteMapping("/{book_id}")
  public ResponseEntity<?> deleteBook(@PathVariable("book_id") String bookId){
  }
}
```

11.8. Creating the view

- The home page JSP

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
  <head>
    <title>Spittr</title>
    <link rel="stylesheet" type="text/css" href="<c:url value="/resources/style.css" />" >
```

```
  </head>
  <body>
    <h1>Welcome to Spittr</h1>
    <a href="<c:url value="/spittles" />">Spittles</a> |
    <a href="<c:url value="/spitter/register" />">Register</a>
  </body>
</html>
```

11.9. Testing the controller

- Since the controller is a POJO, it is easy to test

```
public class HomeControllerTest {
  @Test
  public void testHomePage() throws Exception {
    HomeController controller = new HomeController();
    assertEquals("home", controller.home());
  }
}
```

- But this is not a good test...
  - What makes this class a `Controller`?
  - Will the `home()` method be called when a `GET` request for `"/"` comes in?
  - Is `"home"` a `View`?
- Much better!

```
public class HomeControllerTest{
  @Test
  public void testHomePage() throws Exception {
    HomeController controller = new HomeController();
    MockMvc mockMvc = standaloneSetup(controller).build();
    mockMvc.perform(get("/")).andExpect(view().name("home"));
  }
}
```

- This time, we issue a `GET` request and test the resulting view as `"home"`
- We created a `MockMvc` container to test controllers
- No need to fire up a web server or web browser!

11.10. Defining class-level request handling

- You can split up the `@RequestMapping` annotations on class and method
  - The class-level `@RequestMapping` applies to all handler methods of the controller
  - Any `@RequestMapping` annotation on the methods complement the class-level one
  - You can add multiple values as an array of Strings to handle multiple paths with the same controller/method

```
@Controller
@RequestMapping({"/", "/homepage"})
public class HomeController{
  @RequestMapping(method=GET)
  public String home() {
    return "home";
  }
}
```

11.11. Passing model data to the view

- To get model data, you will need a repository

```
public interface SpittleRepository {
  List<Spittle> findSpittles(long max, int count);
}
```

- You will need an implementation of this interface that fetches data from a database
- The model bean is just a POJO data object

```
public class Spittle {
  private final Long id;
  private final String message;
  private final Date time;
  private Double latitude;
  private Double longitude;
  // constructors, getters, setters, equals and hashCode
}
```

- Creating the controller

```
@Controller
@RequestMapping("/spittles")
public class SpittleController {
  private SpittleRepository spittleRepository;

  @Autowired
  public SpittleController(SpittleRepository spittleRepository) { // inject repository
    this.spittleRepository = spittleRepository;
  }

  @RequestMapping(method=RequestMethod.GET)
  public String spittles(Model model) {
    // adding to the model
    model.addAttribute(spittleRepository.findSpittles(Long.MAX_VALUE, 20));
    return "spittles"; // returning the view name
  }
}
```

- Notice the `Model` parameter...
- Testing the controller

```
@Test
public void shouldShowRecentSpittles() throws Exception {
  List<Spittle> expectedSpittles = createSpittleList(20);
  SpittleRepository mockRepository= mock(SpittleRepository.class);
  when(mockRepository.findSpittles(Long.MAX_VALUE, 20)).thenReturn(expectedSpittles);
  SpittleController controller = new SpittleController(mockRepository);
  // setSingleView() is necessary to not confuse the view path with the request path
  MockMvc mockMvc = standaloneSetup(controller)
    .setSingleView(new InternalResourceView("/WEB-INF/views/spittles.jsp"))
    .build();
  // see next block
}
```

- Testing the controller (continued)

```
// insert in previous block
mockMvc.perform(get("/spittles"))
  .andExpect(view().name("spittles"))
  .andExpect(model().attributeExists("spittleList"))
  .andExpect(model().attribute("spittleList", hasItems(expectedSpittles.toArray())));
```

```
// ...
private List<Spittle> createSpittleList(int count) {
  List<Spittle> spittles = new ArrayList<Spittle>();
  for (int i = 0; i < count; i++) {
```

```
    spittles.add(new Spittle("Spittle " + i, new Date()));
  }
  return spittles;
}
```

- The `Model` parameter is essentially a map
  - It will be handed over to the `View` so the data can be rendered to the client
  - The name of the key is inferred from the type of object being set as value
    - In case of a `List<Spittle>`, the key will be inferred as `"spittleList"`
  - You can specify the key name explicitly, and you can replace `Model` with `Map` if you prefer to work with a non-Spring type
- Spring can make the controller method shorter
  - This method returns the model data, which is put into the model
  - The model key is inferred from the type, in this case `"spittleList"`
  - The logical view name is derived from the path, so it becomes `"spittles"`

```
@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles() {
  return spittleRepository.findSpittles(Long.MAX_VALUE, 20));
}
```

- The model data is copied into the request as request attributes
- In `/WEB-INF/views/spittles.jsp`, we can access the model data with JSTL

```
<c:forEach items="${spittleList}" var="spittle">
  <li id="spittle_<c:out value="spittle.id"/>">
    <div class="spittleMessage">
      <c:out value="${spittle.message}" />
    </div>
    <div>
      <span class="spittleTime"><c:out value="${spittle.time}" /></span>
      <span class="spittleLocation">
        (<c:out value="${spittle.latitude}" />,<c:out value="${spittle.longitude}"/>)
      </span>
    </div>
  </li>
</c:forEach>
```

11.12. Accepting request input

- Clients can send data back to the server
- Without this feature, the web would be read-only
- Spring MVC accepts
  - Query parameters
  - Form parameters
  - Path variables
- Taking query parameters

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

- You can add default values for the case when the parameters are not present

```
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);
}
```

- Taking path variables
  - Instead of `/spittles/show?spittle_id=12345`, prefer `/spittles/12345`
  - This identifies a resource, following REST principles
  - Notice that you can omit the value parameter in `@PathVariable` because the method parameter name is the same as the placeholder name

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

```
<div class="spittleView">
  <div class="spittleMessage"><c:out value="${spittle.message}" /></div>
  <div>
    <span class="spittleTime"><c:out value="${spittle.time}" /></span>
  </div>
</div>
```

- Processing forms
  - There are two sides to working with forms: displaying the form and processing the data the user submits from the form
- Displaying the form
  - Renders `/WEB-INF/views/registerForm.jsp` when requesting `/spitter/register`

```
@Controller
@RequestMapping("/spitter")
public class SpitterController {
  @RequestMapping(value="/register", method=GET)
  public String showRegistrationForm() {
    return "registerForm";
  }
}
```

- Displaying the form

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
  <head>
    <title>Spittr</title>
    <link rel="stylesheet" type="text/css" href="<c:url value='/resources/style.css' />" >
  </head>
  <body>
    <h1>Register</h1>
    <form method="POST"> <!--no action parameter posts back to /spitter/register -->
      First Name: <input type="text" name="firstName" /><br/>
      Last Name: <input type="text" name="lastName" /><br/>
      Username: <input type="text" name="username" /><br/>
      Password: <input type="password" name="password" /><br/>
      <input type="submit" value="Register" />
    </form>
  </body>
</html>
```

- Writing form handling

```
@Controller
@RequestMapping("/spitter")
public class SpitterController{
  private SpitterRepository spitterRepository;

  @Autowired
  public SpitterController(SpitterRepository spitterRepository) {
    this.spitterRepository = spitterRepository;
  }

  @RequestMapping(value="/register", method=GET)
  public String showRegistrationForm() {
    return "registerForm";
  }

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

- Writing form handling
  - Notice the `processRegistration()` method is given an `Spitter` object
    - This object will be populated from the request parameters of the same name
  - Rather than returning the `View`, we return a `redirect` specification
    - Instead of a view name, the browser will receive a redirect for the user's profile page
  - Next to `redirect` you can also use `forward` which will forward the request to the given URL
- Adding a handler for the redirected page

```
@RequestMapping(value="/{username}", method=GET)
public String showSpitterProfile(@PathVariable String username, Model model) {
  Spitter spitter = spitterRepository.findByUsername(username);
  model.addAttribute(spitter);
  return "profile";
}
```

11.13. Validating forms

- Spring supports the Java Validation API (JSR-303)
  - No extra configuration is necessary, just add Hibernate Validator on the classpath
- The Java Validation API specifies several annotations you can put on properties, but you can add your own!
  - `@AssertFalse`, `@AssertTrue`
  - `@DecimalMin`, `@DecimalMax`
  - `@Digits`
  - `@Future`, `@Past`
  - `@Min`, `@Max`
  - `@Null`, `@NotNull`
  - `@Pattern`
  - `@Size`
- Adding validation annotations to the `Spitter` bean

```
public class Spitter {
  private Long id;
  @NotNull
  @Size(min=5, max=16)
  private String username;
  @NotNull
```

```
  @Size(min=5, max=25)
  private String password;
  @NotNull
  @Size(min=2, max=30)
  private String firstName;
  @NotNull
  @Size(min=2, max=30)
  private String lastName;
  // ...
}
```

- We can now add a validation check to the controller method
  - Note the `Errors` parameter must follow the `@Valid`-annotated parameter
  - If the form has any errors, we take the user back to the form so they can correct any problems and try again

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(@Valid Spitter spitter, Errors errors) {
  if (errors.hasErrors()) {
    return "registerForm";
  }
  spitterRepository.save(spitter);
  return "redirect:/spitter/" + spitter.getUsername();
}
```

11.14. Rendering web views

- None of the controller methods produce HTML that is rendered in the browser
  - They populate the model with some data
  - They pass the model off to a view for rendering
  - The methods return a String value that is the logical name of the view
  - `Controller`s are not aware of the views and technology used for rendering
- Spring uses `ViewResolver`s to determine the actual view to render
  - Next to the `InternalResourceViewResolver`, Spring provides several out-of-the-box implementations
- In most applications, you only need a handful of these view resolvers
  - For the most part, each resolver corresponds to a specific view technology available for Java web applications

View Resolver	View Resolver
`BeanNameViewResolver`	`UrlBasedViewResolver`
`ContentNegotiatingViewResolver`	`VelocityLayoutViewResolver`
`FreeMarkerViewResolver`	`VelocityViewResolver`
`InternalResourceViewResolver`	`XmlViewResolver`
`JasperReportsViewResolver`	`XsltViewResolver`
`ResourceBundleViewResolver`	`TilesViewResolver`

11.15. Creating JSP views

- Spring supports JSP pages in two ways
  - `InternalResourceViewResolver` and JSTL `Locale` / `ResourceBundle` support
  - Two JSP tag libraries with form-to-model binding and general utilities
- `InternalResourceViewResolver` uses a convention to determine the path to your JSP pages
  - Placing your pages in `WEB-INF` is a common practice to prevent direct access

- By adding a prefix and a suffix, the physical view path can be derived

```
@Bean
public ViewResolver viewResolver() {
  InternalResourceViewResolver resolver = new InternalResourceViewResolver();
  resolver.setPrefix("/WEB-INF/views/");
  resolver.setSuffix(".jsp");
  return resolver;
}
```

11.16. Resolving JSTL views

- If your JSP pages are using JSTL, you may want to configure the `InternalResourceViewResolver` to resolve a `JstlView`
  - JSTL tags will then be given a `Locale` and a message source configured in Spring
  - This is useful for properly formatting dates and money

```
@Bean
public ViewResolver viewResolver() {
  InternalResourceViewResolver resolver = new InternalResourceViewResolver();
  resolver.setPrefix("/WEB-INF/views/");
  resolver.setSuffix(".jsp");
  resolver.setViewClass(org.springframework.web.servlet.view.JstlView.class);
  return resolver;
}
```

11.17. Binding forms to the model

- Tag libraries bring functionality to a JSP template without needing Java code directly in scriptlet blocks
- The Spring form-binding JSP tag library contains 14 tags
  - These are used to render HTML `form` tags
  - These tags can be bound to an object in the model and can be populated with values from the model's object properties
- To use the form-binding tag library, you will need to declare it
  - Notice the prefix `sf` for Spring forms
  - It is common to find a prefix of `form`

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="sf" %>
```

- These are the 14 tags of the form-binding tag library

JSP Tag	JSP Tag
<sf:checkbox>	<sf:option>
<sf:checkboxes>	<sf:options>
<sf:errors>	<sf:password>
<sf:form>	<sf:radiobutton>
<sf:hidden>	<sf:radiobuttons>
<sf:input>	<sf:select>
<sf:label>	<sf:textarea>

- Applying those tags on the registration form

```
<sf:form method="POST" commandName="spitter">
  First Name: <sf:input path="firstName" /><br/>
```

```
  Last Name: <sf:input path="lastName" /><br/>
  Email: <sf:input path="email" /><br/>
  Username: <sf:input path="username" /><br/>
  Password: <sf:password path="password" /><br/>
  <input type="submit" value="Register" />
</sf:form>
```

- Notice the `commandName` attribute set to `spitter`

```
@RequestMapping(value="/register", method=GET)
public String showRegistrationForm(Model model) {
  model.addAttribute(new Spitter());
  return "registerForm";
}
```

- The `<sf:input>` tag allows you to specify a `type` attribute so that you can declare HTML5 specific text fields, such as `date`, `range`, `email` ...

```
Email: <sf:input path="email" type="email" /><br/>
```

- To guide the user when they make errors, we need to add the `<sf:errors>` tag

```
<sf:form method="POST" commandName="spitter">
  First Name: <sf:input path="firstName" />
  <sf:errors path="firstName" cssClass="error"/><br/>
  <!-- ... -->
</sf:form>
```

- If there is a validation error for the property, the `<sf:errors>` will render the error message in a HTML `<span>` tag
  - You can use the `cssClass` attribute to make the error stand out
- Another way to show errors, is by displaying them all together

```
<sf:form method="POST" commandName="spitter" >
  <sf:errors path="*" element="div" cssClass="errors" />
  <!-- ... -->
</sf:form>
```

- On the `<sf:label>` and `<sf:input>` you can also add a `cssErrorClass` to highlight the fields to be corrected

```
<sf:form method="POST" commandName="spitter" >
  <sf:label path="firstName" cssErrorClass="error">First Name</sf:label>:
  <sf:input path="firstName" cssErrorClass="error" /><br/>
  <!-- ... -->
</sf:form>
```

- You can set the `message` attribute on validation annotations, to make errors friendlier to read and to internationalize them
  - The message will be defined in a properties file named `ValidationMessages.properties`

```
@NotNull
@Size(min=5, max=16, message="{username.size}")
private String username;
@NotNull
@Size(min=5, max=25, message="{password.size}")
private String password;
@NotNull
@Size(min=2, max=30, message="{firstName.size}")
private String firstName;
@NotNull
@Size(min=2, max=30, message="{lastName.size}")
private String lastName;
@NotNull
@Email(message="{email.valid}")
private String email;
```

11.18. Spring's general tag library

- Next to the form-binding tags, Spring offers a more general JSP library
  - Notice the prefix `s` is usually set to `spring`

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
```

- This gives you 10 JSP tags

JSP Tag	JSP Tag
`<s:bind>`	`<s:nestedPath>`
`<s:escapeBody>`	`<s:theme>`
`<s:hasBindErrors>`	`<s:transform>`
`<s:htmlEscape>`	`<s:url>`
`<s:message>`	`<s:eval>`

- Most of these are no longer necessary due to the Spring form-binding tags
  - The `<s:bind>` was Spring's original form-binding tag, but it was much more complex to use
- You can use the `<s:message>` tags for internationalization
- Instead of ...

```
<h1>Welcome to Spittr!</h1>
```

- ... you should use

```
<h1><s:message code="spittr.welcome" /></h1>
```

- Use a `ResourceBundleMessageSource` to load the properties files

```
@Bean
public MessageSource messageSource() {
  ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
  messageSource.setBasename("messages");
  return messageSource;
}
```

- Optionally, a `ReloadableResourceBundleMessageSource` can be configured that allows to reload message properties without restarting the application

```
@Bean
public MessageSource messageSource() {
  ReloadableResourceBundleMessageSource messageSource = new ReloadableResourceBundleMessageSource();
  messageSource.setBasename("file:///etc/spittr/messages"); // looks in the file system
  messageSource.setCacheSeconds(10);
  return messageSource;
}
```

- `<s:url>` creates servlet-context specific URLs

```
<s:url href="/spitter/register" var="registerUrl" /> <!--/spittr/spitter/register -->
<a href="${registerUrl}">Register</a>
```

- With `<s:param>` parameters can be added
  - You can optionally escape the URL for HTML and JavaScript use

```
<s:url href="/spittles" var="spittlesUrl" htmlEscape="true" javaScriptEscape="true">
  <s:param name="max" value="60" />
  <s:param name="count" value="20" />
</s:url>
```

- The `<s:escapeBody>` is a general purpose escaping tag

```
<s:escapeBody htmlEscape="true">
  <h1>Hello</h1> <!--rendered as &lt;h1&gt;Hello&lt;/h1&gt; -->
</s:escapeBody>
```

11.19. Working with Thymeleaf

- Thymeleaf is a templating library
- Thymeleaf templates
  - Are natural, easy to write, and do not rely on tag libraries
  - Can be edited and rendered anywhere raw HTML is welcome
  - Are not tied to the Servlet specification
- To configure Thymeleaf, you need three beans
  - `ThymeleafViewResolver`
  - `SpringTemplateEngine`
  - `TemplateResolver`

```
@Bean
public ViewResolver viewResolver(SpringTemplateEngine templateEngine) {
  ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
  viewResolver.setTemplateEngine(templateEngine);
  return viewResolver;
}

@Bean
public TemplateEngine templateEngine(TemplateResolver templateResolver) {
  SpringTemplateEngine templateEngine = new SpringTemplateEngine();
  templateEngine.setTemplateResolver(templateResolver);
  return templateEngine;
}

@Bean
public TemplateResolver templateResolver() {
  TemplateResolver templateResolver = new ServletContextTemplateResolver();
  templateResolver.setPrefix("/WEB-INF/templates/");
  templateResolver.setSuffix(".html");
  templateResolver.setTemplateMode("HTML5");
  return templateResolver;
}
```

11.20. Defining Thymeleaf templates

- Thymeleaf templates are essentially just HTML files
  - It adds its attributes to the standard set of HTML tags via a custom namespace
  - The `th:href` attribute value can contain Thymeleaf expressions to evaluate dynamic values

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Spittr</title>
    <link rel="stylesheet" type="text/css" th:href="@{/resources/style.css}"></link>
  </head>
  <body>
    <h1>Welcome to Spittr</h1>
    <a th:href="@{/spittles}">Spittles</a> |
```

```
    <a th:href="@{/spitter/register}">Register</a>
  </body>
</html>
```

- As the template is just HTML, it renders nicely in browsers/tools

11.21. Form binding with Thymeleaf

- Consider the following snippet

```
<label th:class="${#fields.hasErrors('firstName')} ? 'error'">First Name</label>:
<input type="text" th:field="*{firstName}" th:class="${#fields.hasErrors('firstName')} ? 'error'" />
```

- `<th:class>` renders a class attribute
- `<th:field>` references the field from the backing object
  - This gives you both a `name` and `value` attribute set to `firstName`
- The `${}` are variable expressions; in Spring they're SpEL expressions
- The `*{}` are selection expressions, evaluated on the selected object
  - In this case the `spitter` object
- The complete form

```
<form method="POST" th:object="${spitter}">
  <div class="errors" th:if="${#fields.hasErrors('*')}">
    <ul>
      <li th:each="err : ${#fields.errors('*')}" th:text="${err}">Input is incorrect</li>
    </ul>
  </div>
  <label th:class="${#fields.hasErrors('firstName')}? 'error'"> First Name</label>:
  <input type="text" th:field="*{firstName}"
    th:class="${#fields.hasErrors('firstName')}?'error'" /><br/>
  <label th:class="${#fields.hasErrors('lastName')}? 'error'"> Last Name</label>:
  <input type="text" th:field="*{lastName}" th:class="${#fields.hasErrors('lastName')}?'error'" /><br/>
  <label th:class="${#fields.hasErrors('email')}? 'error'"> Email</label>:
  <input type="text" th:field="*{email}" th:class="${#fields.hasErrors('email')}? 'error'"/><br/>
  <!-- ... -->
```

- The complete form (continued)

```
  <!-- ... -->
  <label th:class="${#fields.hasErrors('username')}? 'error'">Username</label>:
  <input type="text" th:field="*{username}" th:class="${#fields.hasErrors('username')}?'error'" /><br/>
  <label th:class="${#fields.hasErrors('password')}? 'error'"> Password</label>:
  <input type="password" th:field="*{password}"
    th:class="${#fields.hasErrors('password')}?'error'" /><br/>
  <input type="submit" value="Register" />
</form>
```

- Notice
  - Error rendering at the top of the form
  - The `th:each` that allows to loop, `th:if` that specifies conditional output
  - The `th:object` attribute referring to the `Spitter` object from the model
  - The use of `${}` and `*{}`

11.22. Exercise: Spring Web MVC

- Explore the MVC project offered by the trainer
- Notice the Thymeleaf pages, the `domain`, `dao` and `config` packages

- Create `Controller`s to make the web applications work completely
  - Add a `HomeController` that refers to the index page
  - Add an `AuthorController` that lists out the list of authors
  - Add a `RegistrationController` that lets you register new `Author`s with a form
- Edit the Thymeleaf pages to add the missing elements and links
- Test your MVC exercise in the browser, so that everything works!

12. Spring Test

12.1. Test-driven development with Spring

- Because Spring takes Dependency Injection to heart, your code becomes suitable for unit testing
  - You can easily create mock dependencies and set them in the object you need to test
  - This allows you to focus on the class or method under test
  - You can remove any environment dependency
- Unit testing is not sufficient
  - At some point, you need to bring all the parts of the application together to see if they will work
  - This is integration testing
- Spring provides first-class testing support
  - Spring will help you write integration tests without deploying and running the whole system
- The Spring `TestContext` Framework is independent of the test framework of choice
  - You can use it while running tests in a standalone environment
  - You can use either JUnit or TestNG to run tests
- Goals
  - Ease of configuration and creation of the Spring Container
  - Injecting dependencies in beans as well as test suites
  - Help with testing database interactions and ORM code in transaction context
  - Test web functionality without deploying the application code in a web container

12.2. Configuring the ApplicationContext

- You can use XML, JavaConfig and automatic wiring with unit tests
- Make sure you have the dependencies for Spring Test in your pom file

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>4.x.x.RELEASE</version> <!-- Select latest release -->
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>4.x.x.RELEASE</version> <!-- Select latest release -->
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

- Start the Spring `TestContext` with the JUnit annotation `@RunWith`
- Use `@ContextConfiguration` to load the configuration

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext.xml")
public class AccountIntegrationTests {
  @Autowired
  private AccountService accountService;

  @Test
  public void accountServiceShouldBeInjected() {
    Assert.assertNotNull(accountService);
  }
}
```

- You can specify multiple configuration files to load
  - The default is to search for `<TestClassName>`-context.xml in the same package as the test
- Loading JavaConfig classes

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={TestConfig.class})
public class AccountIntegrationTestsWithJavaConfig {
  // ...
}
```

- Behind the scenes, different classes handle the work
  - `TestContext`, `TestContextManager`, `TestExecutionListener`, `ContextLoader`, `SmartContextLoader`
  - You don't interact with these classes directly
- You can also combine multiple configurations, XML and JavaConfig, but not on the same class

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={TestConfig.class, ServiceConfig.class})
@ActiveProfiles(profiles={"test","c3p0"})
public class AccountIntegrationTestsWithJavaConfig {
  @Configuration
  @ImportResource("classpath:/applicationContext.xml")
  static class Config {
  }
  // ...
}
```

- Activate the profiles you need with `@ActiveProfiles`

12.3. Inheriting context configuration

- Spring supports inheriting from base test classes

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={BaseTestConfig.class})
public class BaseTest {
  @Autowired
  protected Foo foo;
}
```

```
@ContextConfiguration(classes={ChildTestConfig.class}) // context from BaseTest is inherited
public class ChildTest extends BaseTest {
  @Autowired
  private Bar bar;

  @Test
  public void dependenciesShouldBeAvailable() {
    Assert.assertNotNull(foo);
    Assert.assertNotNull(bar);
  }
}
```

12.4. ApplicationContext caching

- The exact same configuration classes and XML locations can be specified on several test classes
  - In that case Spring creates the `ApplicationContext` only once and shares it among those test classes at runtime
  - The cache is kept in a static variable

- Gotcha!
  - For this to work, test suites need to run in the same process
  - If you run tests, you must make sure the build tool does not fork between tests
- If you need to discard the context, use `@DirtiesContext`

12.5. Injecting dependencies in tests

- You can use `@Autowired`, `@Qualifier`, `@Resource`, `@Inject` and `@Named`

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=DependencyInjectionTestConfig.class)
public class DependencyInjectionTests {
  @Autowired
  @Qualifier("foo1")
  private Foo foo1;

  @Resource
  private Foo foo2;

  @Resource
  private Bar bar;

  @Test
  public void testInjections() {
    Assert.assertNotNull(foo1);
    Assert.assertNotNull(foo2);
    Assert.assertNotNull(bar);
  }
}
```

- If you need programmatic access to the `ApplicationContext`, you can `@Autowire` it in

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class DependencyInjectionTests {
  @Autowired
  private ApplicationContext applicationContext;
  // ...
}
```

12.6. Using transaction management in tests

- You can execute tests within a transactional context
  - You will need a `transactionManager` of type `PlatformTransactionManager`
  - You need to place `@Transactional` on either the class or method

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=TransactionalTestConfig.class)
@Transactional
public class TransactionalTests {
  @Test
  public void transactionalTestMethod1() {
    // ...
  }

  @Test
  public void transactionalTestMethod2() {
    // ...
  }
}
```

- The main philosophy of unit tests is:

Expect a clean environment before you run and leave that environment clean after you finish execution

- In accordance with this philosophy, Spring will rollback the transaction at the end of the test method instead of committing
  - That way, changes in the database will not cause side effects in later tests that also use the database
- If you need the transaction to commit, use `@Rollback(false)`
- JUnit provides `@Before` and `@After` annotations to create fixtures
  - These let you run code snippets before and after the execution of each test method
  - If the method is `@Transactional`, these methods will run within the transaction
- If you want to execute some code outside the transaction, you can use
  - `@BeforeTransaction`
  - `@AfterTransaction`
- Spring will try to find a bean with the exact name `transactionManager`
  - Configure the behavior with `@TransactionConfiguration`

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=TransactionalTestConfig.class)
@Transactional
@TransactionConfiguration(transactionManager="myTxMgr",defaultRollback=false)
public class TransactionalTests {
  // ...
}
```

12.7. Testing with ORM frameworks

- Be careful for caching with ORM frameworks!
  - ORM frameworks accumulate persistence operations in their internal state
  - Those operations are usually executed at transaction commit time
  - Since the Spring framework rolls back the transaction, the persistence operations may not even be executed!
- You will need to flush the `EntityManager` at the end of the test method
  - This will ensure interaction with the database, so you are certain any database constraints are also triggered when needed
- For fast database tests, you can use an embedded database

12.8. Testing web applications

- With Spring `TestContextFramework` you can load a `WebApplicationContext` in an integration test

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=WebApplicationTestConfig.class)
@WebAppConfiguration
public class WebApplicationTests {
  @Autowired
  private WebApplicationContext applicationContext;

  @Autowired
  private MockServletContext servletContext;

  @Test
  public void testWebApp() {
    Assert.assertNotNull(applicationContext);
```

```
    Assert.assertNotNull(servletContext);
  }
}
```

- Behind the scenes, Spring creates a `ServletContext` of type `MockServletContext`
  - By default, its base resource path is the `src/main/webappfolder`
  - Change the path with `@WebApplicationConfiguration`
- Spring also creates the following
  - `MockHttpServletRequest`
  - `MockHttpServletResponse`
- These are created per test method in a test suite and put into the Spring Web's thread local `RequestContextHolder`
  - This is cleared after the test method completes

12.9. Testing request- and session-scoped beans

- `LoginAction` is request-scoped, `UserPreferences` is session-scoped

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(classes=ScopedBeanTestConfig.class)
public class ScopedBeanTests {
  @Autowired
  private UserService userService;

  @Autowired
  private MockHttpServletRequest httpServletRequest;

  @Autowired
  private MockHttpSession httpSession;

  @Test
  public void testScopedBeans() {
    httpServletRequest.setParameter("username", "jdoe");
    httpServletRequest.setParameter("password", "secret");
    httpSession.setAttribute("theme", "blue");
    Assert.assertEquals("jdoe", userService.getLoginAction().getUsername());
    Assert.assertEquals("secret", userService.getLoginAction().getPassword());
    Assert.assertEquals("blue", httpSession.getAttribute("theme"));
  }
}
```

In the code above, the `MockHttpServletRequest` and `MockHttpSession` are created by Spring. Any beans with the corresponding scope will be created and be available for injection. In the test method, we can expect these beans to be available in the `UserService`.

12.10. Testing Spring MVC projects

- You can invoke the `DispatcherServlet` from your test code
  - This enables you to run integration tests without the servlet container!
- Imagine the following controller

```
@Controller
public class HelloController{
  @RequestMapping(value = "/hello")
  public ModelAndView sayHello() {
    ModelAndView mv = new ModelAndView();
    mv.addObject("message", "Hello Kitty!");
    mv.setViewName("hello");
    return mv;
  }
}
```

Notice how we return a `ModelAndView` object from this controller. This corresponds to step 4 of the Spring MVC architecture.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(classes=WebAppConfig.class)
public class HelloControllerTests{
  @Autowired
  private WebApplicationContext wac;

  private MockMvc mockMvc;

  @Before
  public void setup() {
    this.mockMvc= MockMvcBuilders.webAppContextSetup(this.wac).build();
  }

  @Test
  public void helloControllerWorksOk() throws Exception {
    mockMvc.perform(get("/hello"))
      .andExpect(status().isOk())
      .andExpect(model().attribute("message", "Hello Kitty!"))
      .andExpect(view().name("hello"));
  }
}
```

12.11. Testing form submission

- Imagine the following controller

```
@Controller
public class UserController{
  @RequestMapping(value = "/form")
  public ModelAndView user() {
    ModelAndView modelAndView = new ModelAndView("userForm", "user", new User());
    return modelAndView;
  }

  @RequestMapping(value = "/result", method=POST)
  public ModelAndView processUser(@Valid User user, Errors errors) {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject("user", user);
    if (errors.hasErrors()) {
      modelAndView.setViewName("userForm");
    } else {
```

```
      modelAndView.setViewName("userResult");
    }
    return modelAndView;
  }
}
```

- First test the happy flow of the form

```
@Test
public void formSubmittedSuccessfully() throws Exception {
  this.mockMvc.perform(post("/result")
      .param("username", "johndoe")
      .param("email", "john@doe.com"))
    .andExpect(status().isOk())
    .andExpect(view().name("userResult"))
    .andExpect(model().hasNoErrors())
    .andExpect(model().attribute("user", hasProperty("username", is("johndoe"))))
    .andExpect(model().attribute("user", hasProperty("email", is("john@doe.com"))));
}
```

- You will need a dependency to `hamcrest-all` to use the `hasProperty()` matcher
- What if a validation error occurs?

```
@Test
public void formSubmittedSuccessfullyButContainsValidationErrors() throws Exception {
  this.mockMvc.perform(post("/result")
      .param("email", "john.doe.com")) // not a valid email and username is missing
    .andDo(print())
    .andExpect(status().isOk())
    .andExpect(view().name("userForm"))
    .andExpect(model().hasErrors());
}
```

- The operation `andDo(print())` makes it possible to print the contents of `MockHttpServletRequest` and `MockHttpServletResponse`
- You may need the following dependencies

```
<dependency>
  <groupId>javax.el</groupId>
  <artifactId>javax.el-api</artifactId>
  <version>3.0.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.glassfish</groupId>
  <artifactId>javax.el</artifactId>
  <version>3.0.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
```

12.12. Testing exception handlers

- What if the controller throws exceptions?

```
@Controller
public class UserController {
  // ...
```

```
  @RequestMapping(value = "/result")
  public ModelAndView processUser(String name) throws Exception {
    ModelAndView modelAndView = new ModelAndView();
    User user = users.get(name);
    if (user == null) { throw new UserNotFoundException(name);}
    modelAndView.addObject("user", user);
    modelAndView.setViewName("userResult");
    return modelAndView;
  }
  @ExceptionHandler
  public ModelAndView handleException(UserNotFoundException e) {
    ModelAndView modelAndView = new ModelAndView("errorUser");
    modelAndView.addObject("errorMessage", e.getMessage());
    return modelAndView;
  }
}
```

- You can then check if the `ExceptionHandler` caught the exception correctly

```
@Test
public void userNotFoundExceptionHandledSuccessfully() throws Exception {
  this.mockMvc.perform(get("/result").param("name", "johndoe"))
  .andExpect(status().isOk())
  .andExpect(view().name("errorUser"))
  .andExpect(model().attribute("errorMessage","User not found with name: johndoe"));
}
```

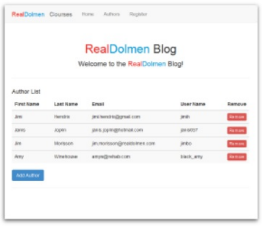
12.13. Spring-provided mock objects and other utilities

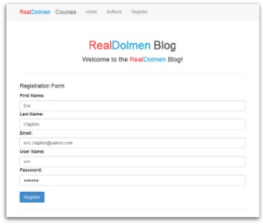
- The `org.springframework.mock` package and subpackages contain more mock object implementations
  - `MockEnvironment`
  - `MockPropertySource`
  - `SimpleNamingContextBuilder`
- Next to these Spring adds several utilities useful for testing
  - `ReflectionTestUtils`
  - `JdbcTestUtils`
- Some annotations can also help specific testing scenarios
  - `@Timed`
  - `@Repeat`
  - `@IfProfileValue`

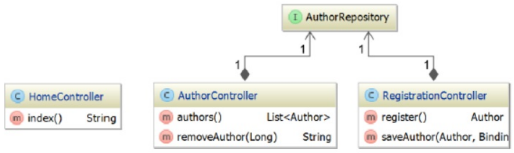
`@Repeat` can be used to quickly load test an application by performing the same test multiple times. Combine it with JUnit parameterized tests to perform the test multiple times on different data sets or with some random function. The scope of the `@Repeat` also includes any set up and tear down of the test fixture.

12.14. Exercise: Spring Test

- Open exercise "test" provided by the teacher
  - In this exercise we will add some unit tests for a small web application
  - Explore the code and run the `Blog` class
    - Navigate to http://localhost:8080







- Exercise one: testing the `HomeController`
  - Our goal is to write a unit test that:
    1. Simulates a `GET` to `/index`
    2. Asserts that it returns status code `200 OK`
    3. Asserts that it yields view name `"index"`
    4. Asserts that the content of the page contains `"This is the homepage of the Realdolmen Blog."`
  - You will need to perform multiple tasks for this:
    1. Enable JUnit to run with Spring's Runner
    2. Spring uses `BlogApplication` as the configuration set with `"test"` as the active profile
    3. A `MockMvc` is created for the entire web app context
- Exercise two: configuring the `TestContext` framework
  - The goal of this exercise is to create a reusable configuration for testing controllers
  - The following steps are needed for this:
    1. Configure `AbstractControllerTest` as a typical Spring test that loads the `BlogApplication` context with `"test"` as the active profile
    2. Create a field for a Mockito `Mock` for `AuthorRepository`
    3. Create a `@Before` fixture to setup a standalone `MockMvc` for the `AuthorController` and `RegistrationController`
    4. Configure both controllers to use the `AuthorRepository` mock
- Exercise three: testing the `Controller`s
  - The goal of this exercise is to implement `RegistrationControllerTest` and `AuthorControllerTest`
  - The following steps are needed for this:
    1. Write a test to verify that invalid input for registration form is not accepted
    2. Write a test to verify that valid input for registration form triggers a database insert
    3. Write a test to verify that `/authors` puts a list of authors on the model
    4. Write a test to verify that `/authors/1/remove` calls the repository's `delete()` method correctly

---

13. Spring Boot

13.1. What is Spring Boot?

- Spring Boot is a framework that helps you get up-and-running with a Spring project quickly
  - Takes an opinionated view of building production-ready applications
  - Favors convention over configuration to minimize configuration
- Features
  - Create stand-alone Spring applications
    - Optionally embedding web container or enterprise features when necessary
  - Provides simplified Maven and Gradle build support
  - Tries to automatically configure the `ApplicationContext`
  - Supports enterprise features such as runtime health checks and other metrics
  - Does not rely on any code generation

13.2. Spring Boot rationale

- Spring is an extremely flexible framework
  - Scalable
    - It scales from the simplest command-line apps to large mission critical enterprise services
  - Extensible
    - It is almost infinitely extensible, allowing you to customize, and fine-tune to your specific needs
- This flexibility does come with a price, however
  - Spring applications do tend to need a high amount of configuration
    - With Spring 4, configuration is already a lot simpler
- Spring Boot is built to allow projects to get started quickly
  - Reducing the initial configuration overhead to an absolute minimum

13.3. Main components

- Spring Boot starters
  - Group typical-use-case dependencies together as a single artifact
    - Supports both Maven and Gradle build systems
- Auto configuration
  - Automatically enable and configure Spring features based on project dependencies
- Command-line interface (CLI)
  - Provide a groovy-based console environment to quickly perform common tasks
- Actuator
  - Add management features typically needed for production grade applications

13.4. Starter dependencies

- Dependency management of a typical Spring app easily becomes unwieldy
  - Gathering the right dependencies
  - Selecting the correct versions to make sure they play together
  - Maintaining (upgrading, adding, removing, ...) these dependencies

- Both Maven and Gradle have this problem
  - Gradle is a lot less verbose than Maven though
- Spring Boot provides a large number of predefined combinations of libraries that eliminate this dependency hell
  - When your project grows larger and more specific dependencies become necessary, they can still be added the usual way
- Consider the Maven dependencies for a simple Spring based web app

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>4.x.x.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.x.x.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>4.x.x.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>4.x.x.RELEASE</version>
</dependency>
```

```
<dependency>
  <groupId>c.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.x.x</version>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>1.4.x</version>
</dependency>
<dependency>
  <groupId>org.thymeleaf</groupId>
  <artifactId>thymeleaf-spring4</artifactId>
  <version>2.x.x.RELEASE</version>
</dependency>
```

- Using Spring Boot's aggregated dependencies, we can shorten this by an order of magnitude
  - Starter dependencies group many other dependencies
  - Framework versions are selected automatically

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.6.RELEASE</version>
</parent>
```

```
<dependency>
  <groupId>o.s.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>o.s.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.thymeleaf</groupId>
  <artifactId>thymeleaf-spring4</artifactId>
</dependency>
```

- The list of available starter dependencies is huge
  - Here is a list of some of the most interesting ones

ArtifactId	ArtifactId	ArtifactId
spring-boot-starter-aop	spring-boot-starter-security	spring-boot-starter-data-mongodb
spring-boot-starter-batch	spring-boot-starter	spring-boot-starter-amqp
spring-boot-starter-data-jpa	spring-boot-starter-test	spring-boot-starter-thymeleaf
spring-boot-starter-data-rest	spring-boot-starter-web	spring-boot-starter-redis
spring-boot-starter-integration	spring-boot-starter-websocket	spring-boot-starter-social-twitter
spring-boot-starter-jdbc	spring-boot-starter-ws	spring-boot-starter-social-linkedin

- GroupId: `org.springframework.boot`

13.5. Automatic configuration

- Automatic configuration simplifies the initial Spring configuration overhead
  - Using classpath scanning, Spring Boot detects which libraries are present
  - When a well-known library is present, Spring Boot automatically configures it using the typical (opinionated) way
- This allows you to start working with the libraries without having to do tedious `ApplicationContext` configuration
- This mechanism uses a "reasonable guess"
  - When its guess is wrong, you can still configure the `ApplicationContext` the usual way
  - You can also configure the default guesses with `application.properties`

13.6. Command-line interface

- Spring Boot uses the power of Groovy to minimize application logic
  - Groovy is a dynamic JVM programming language that gets rid of some of the verbosity of Java
- Any Groovy code can interface with Java, by using some boilerplate logic
  - This boilerplate logic is eliminated by Spring Boot's CLI feature
- Consider the following Groovy script

```
@RestController class Hi { @RequestMapping("/") String hi() { "Hi!" } }
```

- This is a fully functional Spring application! You can run it as follows:

```
C:\project>spring run Hi.groovy
```

13.7. Actuator

- Big enterprise servers typically have a lot of management features
  - Runtime metrics, server health, etc...
- Spring Boot promotes a stand-alone way of running applications

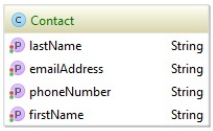
- How can we make sure these stand-alone apps are equally production-grade as the big enterprise suites?

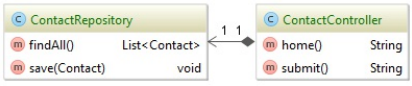
- The Actuator enhances stand-alone applications so that they also offer these features
  - It adds REST endpoints that allow you to query various runtime statistics on your live stand-alone application

13.8. Building a web application

- Using Spring Boot, let us create a simple but representative web application using the following (typical) technologies
  - Maven
  - Spring MVC
  - Thymeleaf (templating)
  - JdbcTemplate
  - H2 (RDBMS)

- As you'll see, as long as you stick to the defaults, configuration is reduced to an absolute minimum
  - You can always change this later, when your requirements become more specific





13.9. Project setup

- First we need to setup the Maven `pom.xml`
  - The project should inherit from `spring-boot-starter-parent`

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.realdolmen.spring</groupId>
  <artifactId>contacts</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.6.RELEASE</version>
  </parent>
</project>
```

- The `pom.xml` should also have the `spring-boot-maven-plugin` plugin
  - This plugin makes sure your application is runnable as a stand-alone jar by repackaging the dependencies into a single "über-jar"
    - Put this under `<build><plugins>`

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

- Since this will be a web application, also add the `spring-boot-starter-web` dependency
  - Put this under `<dependencies>`

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

13.10. Controllers

- Since you have added the web starter dependency, Spring Boot will automatically configure all Spring beans required to enable MVC
  - Adding a Tomcat Servlet engine
  - Setting up the `DispatcherServlet`
  - ...
- This means you can simply start programming the Spring MVC controllers the usual way without any special bean configuration
- Our controller might look like this

```
@Controller
@RequestMapping("/")
public class ContactController {
  @Autowired
  private ContactRepository repository;

  @RequestMapping(method = RequestMethod.GET)
  public String home(Map<String, Object> model) {
    List<Contact> contacts = repository.findAll();
    model.put("contacts", contacts);
    return "home";
  }

  @RequestMapping(method=RequestMethod.POST)
  public String submit(Contact contact) {
    repository.save(contact);
    return "redirect:/";
  }
}
```

13.11. Domain classes

- The domain class is a simple POJO called `Contact`
  - No special Spring Boot mechanics, just a POJO!
    - If you want to use JPA, this domain class can be upgraded to a JPA entity

```
public class Contact {
  private Integer id;
  private String firstName;
  private String lastName;
  private String phoneNumber;
  private String emailAddress;

  // Getters and setters
}
```

13.12. Views

- When using Thymeleaf, we will have to add an extra dependency

```
<dependency>
  <groupId>org.thymeleaf</groupId>
  <artifactId>thymeleaf-spring4</artifactId>
</dependency>
```

- Using the automatic configuration, Spring Boot will automatically enable all beans for Thymeleaf

- `ThymeleafViewResolver`, `SpringTemplateEngine`, `TemplateResolver`, ...

- Simply put the template files in the default location
  - `/src/main/resources/templates`

- The Thymeleaf template might look like this
  - This is a regular template, nothing up the sleeves!

```
<h2>Spring Boot Contacts</h2>
<form method="POST">
  <label for="firstName">First Name:</label><input type="text" name="firstName"/><br/>
  <label for="lastName">Last Name:</label><input type="text" name="lastName"/><br/>
  <label for="phoneNumber">Phone #:</label><input type="text" name="phoneNumber"/><br/>
  <label for="emailAddress">Email:</label><input type="text" name="emailAddress"/><br/>
  <input type="submit"/>
</form>
<ul th:each="contact : ${contacts}">
  <li>
    <span th:text="${contact.firstName}">First</span>
    <span th:text="${contact.lastName}">Last</span> :
    <span th:text="${contact.phoneNumber}">phoneNumber</span>,
    <span th:text="${contact.emailAddress}">emailAddress</span>
  </li>
</ul>
```

13.13. Static resources

- Static resources like JavaScript, CSS and images can be added quickly by sticking to the default locations
  - `src/main/resources/META-INF/resources/`
  - `src/main/resources/resources/`
  - `src/main/resources/static/`
  - `src/main/resources/public/`

- Spring Boot automatically configures a resource handler for these locations
  - For example, we can add a file style.css to any of the above directories

```
body { background-color: #eeeeee; font-family: sans-serif; }
label { display: inline-block; width: 120px; text-align: right; }
```

13.14. Persistence

- To add persistence, we need two dependencies:
  - `spring-boot-starter-jdbc`
    - Adds all the necessary jars for persistence (spring-jdbc, ...)
  - A JDBC driver

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

- Again, Spring Boot will automatically configure the required Spring beans
  - `DataSource`, `JdbcTemplate`, `TransactionManager`, ...

- Exporting the data schema can be done automatically as well
  - Simply add a file called `schema.sql` under `/src/main/resources`

```
create table contacts (
  id identity,
  firstName varchar(30) not null,
  lastName varchar(50) not null,
  phoneNumber varchar(13),
  emailAddress varchar(30)
);
```

- Spring Boot will detect this file and execute it at boot time using the automatically configured database settings
  - Very useful if you are using an embedded database like H2 during development
- We can now start creating a regular repository implementation using `JdbcTemplate`

```
@Repository
public class ContactRepository{
  @Autowired private JdbcTemplate jdbcTemplate;
  public List<Contact> findAll() {
    return jdbcTemplate.query("select id, firstName, lastName, phoneNumber, " +
    "emailAddress from contacts order by lastName", (rs, rowNum) -> {
      Contact contact = new Contact(rs.getInt(1), rs.getString(2), rs.getString(3),
      rs.getString(4), rs.getString(5));
      return contact;
    });
  }
  public void save(Contact contact) {
    jdbcTemplate.update("insert into contacts(firstName, lastName, phoneNumber, " +
    "emailAddress) values(?, ?, ?, ?)", contact.getFirstName(),
    contact.getLastName(), contact.getPhoneNumber(), contact.getEmailAddress());
  }
}
```

13.15. Application configuration

- One final piece remains: bootstrapping the application
  - Spring Boot applications can run simply as a `main()` application
  - `SpringApplication.run()` instantiates `Main` and treats it as a `JavaBean` configuration
  - `@SpringBootApplication` is a shorthand for
    - `@EnableAutoconfiguration` (which triggers the automatic bean configurations)
    - `@Configuration`
    - `@ComponentScan` (from the current package and subpackages)
  - No need to configure the `DispatcherServlet`, add a `web.xml`, ...

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

13.16. Running the application

- The application can be launched from the command line
  - This will launch an embedded web server
    - http://locahost:8080/
  - During development this is extremely useful

```
C:\project>mvn package
C:\project>java -jar target/contacts-1.0.jar
```

- Production environments usually still need to deploy the application as a WAR though
  - To do this, simply change the `pom.xml` packaging from `jar` to `war`
    - The application will then be both executable from the command-line as deployable as a WAR

13.17. Enabling the Actuator

- Spring Boot allows you to add a number of REST endpoints to enable server monitoring
- This must be enabled by adding the actuator dependency
  - Using Maven by adding it as a `<dependency>`

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

- Automatic configuration will do the REST!

13.18. Actuator endpoints

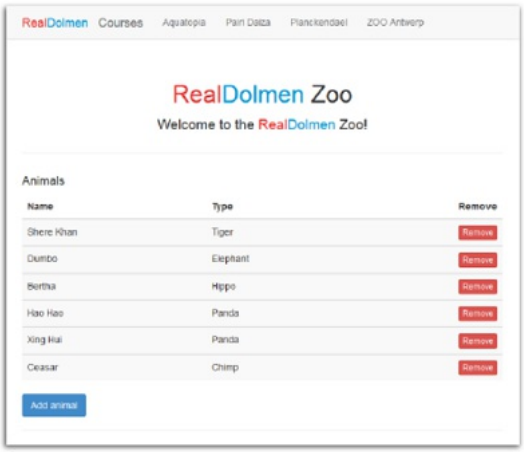
Endpoint	Description
`GET /autoconfig`	Displays auto-configuration guesses
`GET /beans`	Shows the beans in the `ApplicationContext`
`GET /dump`	Shows the application threads with their current stack
`GET /env`	Shows environment properties
`GET /health`	Displays health of the application
`GET /info`	Shows some application info
`GET /metrics`	Shows application statistics such as requests per controller
`POST /shutdown`	Shuts down the server
`GET /trace`	Shows HTTP details about the most recent requests

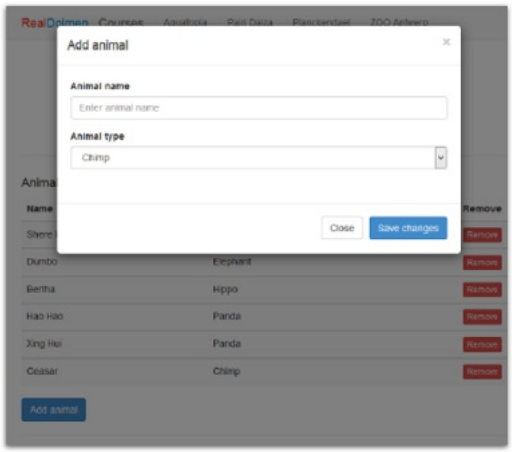
- The actuator endpoints show their information in JSON
  - This can be used to write a client against

```
[{
  "context": "application",
  "parent": null,
  "beans": [{
    "bean": "contactController",
    "scope": "singleton",
    "type": "ContactController",
    "resource": "null",
    "dependencies": ["contactRepository"]
    }, {
    "bean": "contact",
    "scope": "singleton",
    "type": "Contact",
    "resource": "null",
    "dependencies": []
    }]
}]
```

13.19. Exercise: Spring Boot

- Open the "boot" exercise provided by the teacher
  - We will use Spring Boot to build an administration web app for adding and removing animals from the "RealDolmen Zoo"





- Exercise one: setting up a Spring Boot project
  - The goal of this exercise is to configure a Maven Spring Boot project
  - The following tasks are needed for this:
    1. Add the Spring Boot parent to the `pom.xml`
    2. Add the Spring starter dependency for a web application
    3. Add the Spring starter dependency for JDBC
    4. Add Thymeleaf as an explicit dependency
    5. Add H2 as an explicit dependency
    6. Add the Spring Boot plugin
  - This should give us a working project setup!
- Exercise two: configuring the application
  - The purpose of this exercise is to activate the existing code using Spring Boot only
    - A number of classes are already provided; writing controllers and repositories is not the purpose of this exercise
  - The following tasks are needed for this:
    1. Create a `main()` that bootstraps the application using Spring Boot annotations and utilities
    2. Put the provided resources in the correct location
    3. Configure the provided controller and repository as Spring Beans
    4. Wire in the `JdbcTemplate` in the repository (where does it come from?)
    5. Launch the application and make sure everything works!
  - During this exercise, do not use any of the following annotations
    - `@EnableMvc`, `@Configuration`, `@ComponentScan`, `@Import`, `@Bean`
- Exercise three: adding the Actuator
  - The goal of this exercise is to enable the Actuator and try it
  - The following tasks are needed for this:
    1. Enable the Actuator in the project
    2. Boot the application and verify in the logs if the actuator endpoints are activated
    3. Try out the `/health`, `/info` and `/beans` endpoints

14. Spring Boot with Groovy

14.1. Groovy and Spring Boot CLI

- Groovy is a popular alternative JVM language
  - It is fully JVM and Java compatible
    - Very easy to learn when you already know Java (superset)
  - It works with any existing Java code
    - All your existing Java code still works
  - It has much reduced ceremony compared to Java
    - Optional semicolon, default public, convenient operators, dynamic typing, ...
- Spring Boot CLI supports creating Spring applications using Groovy
  - The command-line interface allows you to run a Groovy script without any hassle
    - No bootstrapping logic (main)
    - No initial context configuration
    - No Maven / Gradle build configuration

14.2. Installing the CLI

- Spring Boot CLI needs to be installed first
  - Download and extract it from the Spring webpages
    - http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-installing-the-cli
  - Add the /bin folder to the `PATH` environment variable
  - Or using Windows' environment configuration settings
  - Test it by running the `spring` command from a terminal

```
export SPRING_BOOT_CLI_HOME=<pathtospring boot cli>
export PATH=$SPRING_BOOT_CLI/bin:$PATH
```

```
$ spring --version
```

14.3. Groovy domain classes

- In Groovy, beans are much easier to write
  - Public is the default
  - Properties have getters and setters by default
  - Semicolons are optional
- The previous `Contact` domain class can be written in Groovy as follows
  - Name it `Contact.groovy` in the root of your project (not `src/main/java`)

```
class Contact {
  long id
  String firstName
  String lastName
  String phoneNumber
  String emailAddress
}
```

14.4. Auto-importing

- Groovy applications using Spring Boot CLI make use of auto-importing, as well as auto-configuration
  - You do not need to write any `import` statements for Spring dependencies
  - Groovy does not need imports for many Java packages
    - `java.lang`, `java.math`, `java.util`, `java.io`, `groovy.lang`, `groovy.util`, ...
- Dependencies will be automatically downloaded and imported at runtime
  - Running the code the first time will fail (`NoClassDefFoundError`)
  - Spring Boot will automatically download missing dependencies and load the classes
  - The code is executed a second time automatically

14.5. Grabbing

- Some dependencies can not be auto-imported
  - They are opt in: there is no explicit compile-time dependency to them
  - The auto-configuration mechanism guesses bean configuration based on them existing on the classpath
  - These are the libraries we had to explicitly add using Maven before as well
    - Examples: Thymeleaf, H2, ...
- You can add these dependencies to your Spring Boot CLI application using the `@Grab` annotation
  - Simply add this to the top of any .groovy file that needs it

```
@Grab("h2")
@Grab("thymeleaf-spring4")
```

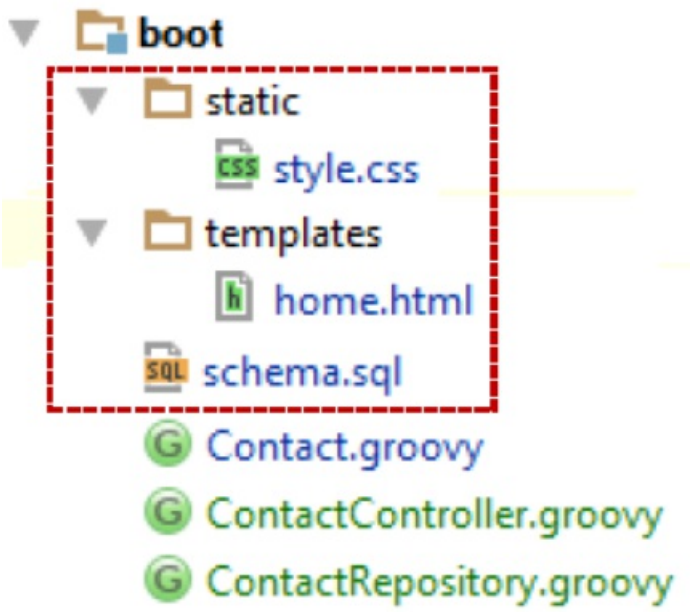
14.6. Groovy controllers

- The controller rewritten in Groovy looks like this
  - Name it `ContactController.groovy` in the root of the project

```
@Grab("thymeleaf-spring4") // No imports!
@Controller
@RequestMapping("/")
class ContactController{
  @Autowired ContactRepository contactRepo
  @RequestMapping(method=RequestMethod.GET)
  String home(Map<String, Object> model) {
    List<Contact> contacts = contactRepo.findAll()
    model.putAll([contacts: contacts]) // Ultra cool map literals!
    "home" // Implicit return!
  }
  @RequestMapping(method=RequestMethod.POST)
  String submit(Contact contact) {
    contactRepo.save(contact)
    "redirect:/"
  }
}
```

14.7. Groovy views and static resources

- Views and static resources can be reused as-is
  - They simply need to be placed directly in the root of the project



14.8. Groovy persistence

- The repository can be rewritten in Groovy as follows
  - Name it `ContactRepository.groovy` in the root of the project

```
@Grab("h2")
import java.sql.ResultSet // Some non-Spring imports still necessary :(
class ContactRepository {
  @Autowired JdbcTemplate jdbc
  List<Contact> findAll() {
    jdbc.query("select id, firstName, lastName, phoneNumber, emailAddress from contacts" +
    "order bylastName", (RowMapper<Contact>){ ResultSetrs, int rowNum ->
      new Contact(id: rs.getLong(1),firstName: rs.getString(2),lastName:
      rs.getString(3),phoneNumber: rs.getString(4),emailAddress: rs.getString(5))})
  }
  void save(Contact c) {
    jdbc.update("insert into contacts(firstName, lastName, phoneNumber, emailAddress)"
    + "values(?, ?, ?, ?)", c.firstName, c.lastName, c.phoneNumber, c.emailAddress)
  }
}
```

14.9. Running from the CLI

- A Groovy based Spring Boot application can be run from the command-line simply by running the `spring run` command
  - All dependencies will be auto-imported or grabbed
  - Spring context is auto-configured
  - Application is launched with the auto-configured `ApplicationContext`
  - Embedded servlet engine is launched at http://localhost:8080

```
$ spring run **/*.groovy
```

- You can still create and run it as a regular stand-alone application as follows

```
$ spring jar myapp.jar **/*.groovy
$ java -jar myapp.jar
```

- Enable the Actuator with Groovy by grabbing it:

```
@Grab("spring-boot-starter-actuator")
```

- Auto-configuration will do the REST!

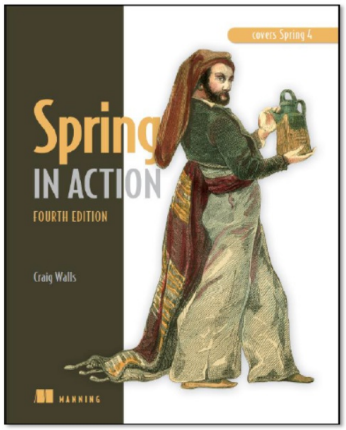
14.10. Exercise: Spring Boot with Groovy

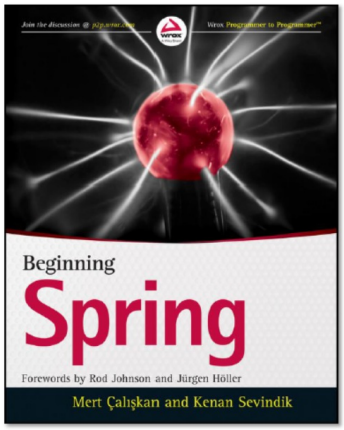
- Optional exercise: using Groovy and the Spring Boot CLI
  - The goal of this exercise is to convert the Spring Boot application to Groovy and run it on the command-line interface
  - The following tasks are needed for this:
    1. Convert your Java classes to Groovy
    2. "Grab" the Thymeleaf and H2 dependencies
    3. Move the groovy and resource files to the project's root
    4. Remove the Maven `pom.xml`, `src/main/java` and `src/main/resources`
    5. Launch the program using the Spring Boot CLI
  - Some hints to convert classes from Java to Groovy:
    - Syntactically almost 100% Java compatible, optional semicolons, map literals: [key: value], no need to import typical Java packages (`java.util`, `java.io`, `java.lang`, `java.math`, ...), public is default, getters and setters are implicit!
    - Yes it's really cool!

15. References

15.1. References

- Books





- Online
  - http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/
  - http://docs.spring.io/spring-boot/docs/
  - http://docs.spring.io/spring-data/jpa/docs/
  - http://www.thymeleaf.org/documentation.html
  - and many more Google searches ...