**Lesson 2. Building a Spring Boot Web App.**

Book and authors example code (Many to Many):

Book:

@ManyToMany  
@JoinTable(name = "author\_book", joinColumns = @JoinColumn(name="book\_id"),  
 inverseJoinColumns = @JoinColumn(name = "author\_id"))  
private Set<Author> authors = new HashSet<>();

Author:

@ManyToMany(mappedBy = "authors")  
private Set<Book> books = new HashSet<>();

Do not forget ID column:

@Id  
@GeneratedValue(strategy = GenerationType.*AUTO*)  
private Long id;

Implement equal and hash

Spring data repositories:

* Provides an implementation of the repository pattern
* Concept is originally from Eric Evans’ book Domain Driven Design
* A Repository has methods for retrieving domain objects should delegate a specialized Repository object such that alternative storage implementation may be interchanged
* This is important. It allows you to easily substitute the persistence layer

**Lesson 3. Dependency Injection with the Spring Framework.**

SOLID OOP principles:

Single Responsibility Principle:

* Every Class should have a single responsibility.
* There should never be more than one reason for a class to change.
* Your classes should be small. No more than a screen full of code.
* Avoid ‘god’ classes.
* Split big classes into smaller classes.

Open/Closed Principle:

* Your classes should be open for extension
* But closed for modification
* You should be able to extend a classes behavior, without modifying it.
* Use private variables with getters and setters - ONLY when you need them.
* Use abstract base classes

Liskov Subsitution Principle:

* By Barbara Liskov, in 1998
* Objects in a program would be replaceable with instances of their subtypes WITHOUT altering the correctness of the program.
* Violations will often fail the “Is a” test.
* A Square “Is a” Rectangle
* However, a Rectangle “Is Not” a Square

Interface Segregation Principle:

* Make fine grained interfaces that are client specific
* Many client specific interfaces are better than one “general purpose” interface
* Keep your components focused and minimize dependencies between them
* Notice relationship to the Single Responsibility Principle?
* ie avoid ‘god’ interfaces

Dependency Inversion Principle:

* Abstractions should not depend upon details
* Details should not depend upon abstractions
* Important that higher level and lower level objects depend on the same abstract interaction
* This is not the same as Dependency Injection - which is how objects obtain dependent objects

**Dependency Injection** is where a needed dependency is injected by another object. The class being injected has no responsibility in instantiating the object being injected.

Types of dependency injection:

* By class properties –least preferred
  + using private properties is EVIL (Impossible to test)
* By setters – area of much debate
* By Constructor – Most preferred

Concrete classes vs interfaces:

* DI can be done with Concrete Classes or with Interfaces
* Generally DI with Concrete Classes should be avoided
* DI via Interface is highly preferred
  + Allows runtime to decide implementation to inject
  + Follows Interface segregation principle of SOLID
  + Makes core more testable

Inversion of control is a technique to allow dependencies to be injected at runtime. Dependencies are not predetermined.

DI refers much to the composition of your classes (you compose your classes with DI in mind). IoC is the runtime environment of your code (Spring Framework’s IoC container)

Use controller as a target and service as a dependency. Use autowired for property and setter based injections. Using autowired for constructor based injection is not mandatory. To get bean from context, first get the context by saving app run function value to a variable and then use getBean method.

Use qualifier annotations to help determine which dependency to use. Method signature example:

public ConstructorInjectedController(@Qualifier("constructorGreetingService") GreetingService greetingService)

With setter you could use @Qualifier in function level or as in constructor.

In property based DI if you use dependency name as a concrete class, then spring will use that concrete class. If @Primary has been set, then dependency will be the class annotated with @Primary annotation.

You could use @Primary technique when you have multiple beans of the same type and you want one of them to go in by default.

**Spring Profiles.** Profiles are something that we can set at runtime which configure Spring how it wires up things so when we set an active profile, Spring is going to bring all beans and then beans that are marked with that Profile and beans of a different profile Spring is going to ignore it. So this is a very important feature we can do things like running multiple data sources if we had two different MYSQL databases or a very common development scenario is to use a primary or development in memory H2 database and MySQL later down the road.

Use @Profile annotation with service. And

spring.profiles.active=es

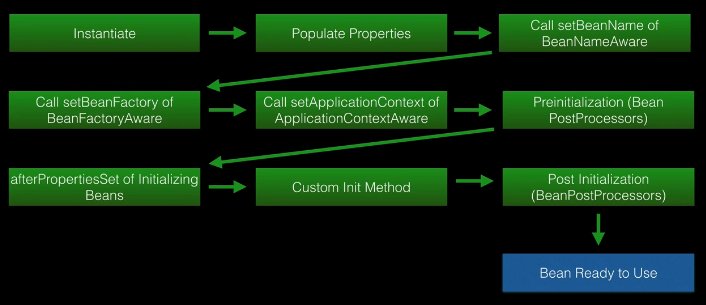
in application properties to set spring profile.

You can specify beans with Default to be added into the Spring context when there are no other beans of that type available so this is a very special case. You could use arrays of profiles:

@Profile({"en", "default"})

Default is only active when no other profiles are active (Only and only. That is important!)

**Spring Bean Life Cycle.**



Callback interfaces. Spring has two interfaces you can implement for call back events:

* InitializingBean.afterPropertiesSet()
  + called after properties are set
* DisposableBean.destroy()
  + called during bean destruction in shutdown

Life Cycle Annotations:

* Spring has two annotations you can use to hook into the bean life cycle
* @PostConstruct annotated methods will be called after the bean has been constructed, but before is returned to the requesting object
* @PreDestroy is called just before the bean is destroyed by the container

Bean Post processors:

* Gives you a mean to tap into the Spring context life cycle and interact with beans as they are processed
* Implement interface BeanPostProcessor
  + postProcessBeforeInitialization – called before bean initialization method
  + postProcessAfterInitialization – called after bean initialization

Aware interfaces:

* Spring has over 14 Aware interfaces
* These are used to access the Spring Framework infrastructure
* These are largely used within the framework
* Rarely used by Spring developers

Release plugin configuration:

<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-maven-plugin</**artifactId**>  
 </**plugin**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-release-plugin</**artifactId**>  
 <**configuration**>  
 <**goals**>install</**goals**>  
 <**autoVersionSubmodules**>true</**autoVersionSubmodules**>  
 <**checkModificationExcludes**>  
 <**checkModificationExclude**>pom.xml</**checkModificationExclude**>  
 </**checkModificationExcludes**>  
 </**configuration**>  
  
 </**plugin**>  
 </**plugins**>  
 </**build**>  
  
 <**scm**>  
 <**developerConnection**>scm:git:https://github.com/avgasanov/sfg-pet-clinic.git</**developerConnection**>  
 <**tag**>HEAD</**tag**>  
</**scm**>

**Lesson 4. Spring Framework Configuration**

* XML Based Configuration
  + Introduced in Spring Framework 2.0
  + Common in legacy Spring Applications
  + Still supported in Spring Framework 5.x
* Annotation Based Configuration
  + Introduced in Spring Framework 3
  + Picked up via ‘Component Scans’
  + Refer to class level annotations
    - @Controller, @Service, @Component, @Repository
* Java Based Configuration
  + Introduced in Spring Framework 3
  + Uses Java Classes to define Spring Beans
  + Configuration classes are defined with @Configuration annotation
  + Beans are declared with @Bean annotation
* Groovy Bean Definition DSL Configuration
  + Introduced in Spring Framework 4
  + Allows you to declare beans in Groovy
  + Borrowed from Grails

Industry trend is to favor Java Based Configuration

Spring Framework Stereotypes

* Stereotype – a fixed general image or set of characteristics which represent a particular type of person or thing
* Spring stereotypes are used to define Spring Beans in the Spring context
* Available Stereotypes - @Component, @Controller, @RestController, @Repository, @Service

Component is the highest level, everything elsee inherits from the Component so when Spring encounters just the Component annotation, that is going to wire it up as a Spring Bean, it is going to hold true for all the other annotations. Controller is used to indicate Spring MVC Controller. @Repository is the annotation that indicates that you are accessing the data layer where Service is intended to be a backing up for that controller. @RestController is a convenience annotation representing @Controller and @ResponseBody

Spring Component scan will work for packages down the package where @SpringBootApplication annotation is defined. Use @ComponentScan(basePackages = {“pack”,…}) for Component scanning in outside packages.

Spring Configuration. Use @Configuration and @Bean to define bean inside configuration class.

Spring Boot Annotations:

* @SpringBootApplication
* @Configuration – Declares class as Spring Configuration
* @EnableAutoConfiguration
* @ComponentScan

Auto-configuration will bring a lot of configuration classes in supplied Spring Boot Jars. You can specify classes to exclude with: @EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})

Spring Bean Scopes:

* Singleton – (default) Only one instance of the bean is created in the IoC container
* Prototype – A new instance is created each time the bean is requested.
* Request – A single instance per http request. Only valid in the context of a web-aware Spring ApplicationContext
* Session – A single instance per http session. Only valid in the context of a web-aware Spring ApplicationContext
* Global-session – A single instance per global session. Typically Only used in a Portlet context. Only valid in the context of a web-aware Spring ApplicationContext.
* Application – bean is scoped to the lifecycle of a ServletContext. Only valied in the context of a web aware.
* Websocket – Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.
* Custom scope – spring scopes are extensible, and you can define your own scope by implementing Spring’s “Scope” interface
* You cannot override the built in Singleton and Prototype Scopes

In Java configuration use @Scope annotation. In XML configuration scope is an XML attribute of the “bean” tag.

To implement Pre loading:

* Create class that implements CommandLineRunner interface
* Add @Component annotation
* implement interface method

**Lesson 5. External Properties with Spring Framework**

Process:

1. Create properties file:
   1. guru.username=John
2. Create Java Configuration class
   1. annotate with @Configuration
   2. and @PropertySource(“classpath:filename.properties”)
   3. add PlaceHolderConfigurer bean:
3. @Bean  
   public static PropertySourcesPlaceholderConfigurer properties() {  
    PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer  
    = new PropertySourcesPlaceholderConfigurer();  
    return propertySourcesPlaceholderConfigurer;  
   }
   1. add fields with @Value(“${property.name}”)
   2. use these fields

To get system variables use Environment (from spring framework)

Use propertysources with string array, or PropertySources with PropertySource array

YAML

You could use – application-de.properties (de – profile name) to use profile specific properties in application. Or you could also use YAML:

--- #new file separator

spring:

profiles: de

#THE rest of the properties