LEARNINGPATTERNS

Presentation Slides:
For INSTRUCTOR Use Only

Introduction to Spring 5 and Spring MVC/REST

Version: 20180521-b

Workshop Overview

- An in-depth course teaching the use of Spring 5 to build enterprise applications using Java
  - Including newer areas of Spring/Java technology
- The course covers the following areas of Spring technology
  - **Core features of Spring**
  - **Spring Boot**
  - **Data Access Features Including Spring Data**
  - **Transaction Support**
  - **Web Application Support including Spring MVC**
  - **Building RESTful Resources with Spring MVC**

Workshop Objectives

- Understand the Spring framework and use its capabilities, including:

- **Spring Core**: Dependency Injection (DI) and bean lifecycle management
  – Spring configuration and API for writing Spring programs
  – XML, Java-based, and annotation-based config

- **Spring Boot**: For easier dependency management and configuration

- **Data Access**: Data access via Spring's data support
  – DataSource support
  – Hibernate and JPA-based Repositories/DAOs
  – Spring Data based repositories

- **Transactions**: Controlling transactions declaratively with Spring
  – Via both Spring annotations and XML configuration

- **Web**: Integrating Spring with Web applications. Understand and use Spring MVC/REST

Workshop Agenda

- Session 1: **Introduction to Spring**
- Session 2: **Configuration in Depth**
- Session 3: **Intro to Spring Boot**
- Session 4: **Spring Testing**
- Session 5: **Database Access With Spring**
- Session 6: **Transactions**
- Session 7: **Web Applications and Spring MVC**
- Session 8: **More Spring MVC Capabilities**
- Session 9: **RESTful Services with Spring**
- Session 10: **Working with JSON and XML**
- Session 11: **Java Clients for RESTful Services**
- Session 12: **Common REST Patterns**
- [Optional] Session 13: **Additional New Features**
- [Optional] Session 14: **XML Specific Configuration**

Typographic Conventions

- Code in the text uses a fixed-width code font, e.g.:

  `Catalog catalog = new CatalogImpl()`

  – Code fragments are the same, e.g. `catalog.speakTruth()`

  – We **bold/color** text for emphasis

  – Filenames are in italics, e.g. *Catalog.java*

  – Notes are indicated with a superscript number [1] or a **star** *

  – Longer code examples appear in a separate code box - e.g.

```
package com.javatunes.teach;

public class CatalogImpl implements Catalog {
   public void speakTruth() {
      System.out.println("BeanFactories are way cool");
   }
}
```

Labs

- The workshop has numerous hands-on lab exercises, structured as a series of brief labs
  - Many follow a common fictional case study called **JavaTunes**
    - An online music store
  - The lab details are separate from the main manual pages
- Setup zip files are provided with skeleton code for the labs
  - Students add code focused on the topic they're working with
  - There is a solution zip with completed lab code
- Lab slides have an icon like in the upper right corner of this slide
  - The end of a lab is marked with a stop like this one:

Session 1: Introduction to Spring

Overview

Spring Introduction

Dependency Injection

Lesson Objectives

- Understand why we need the Spring Framework
- Understand what Spring does, and how it simplifies enterprise application development
- Learn how Spring uses configuration information and Dependency Injection (DI)
  - To manage the beans (objects) in an application
  - To manage bean dependencies

Overview

Overview

Spring Introduction

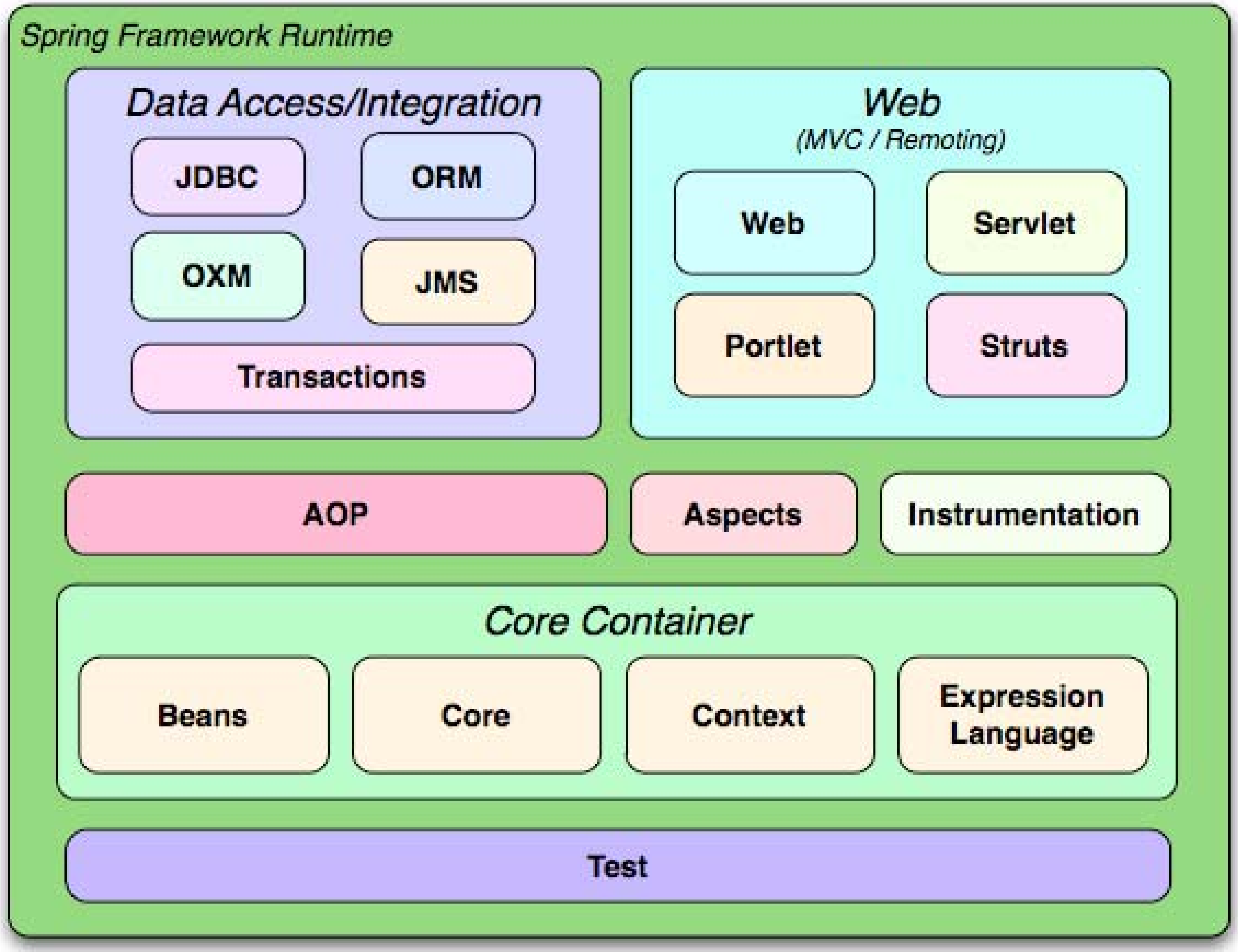
The Spring Container

Dependency Injection

Spring and Enterprise Applications

- Enterprise apps have complex requirements, including
  - Many **application types** and dependencies
  - **Persistent data** and **transactions**
  - **Remote clients** (REST, Web Service, others)
- **Spring: Lightweight framework to build enterprise apps**
  - Non-intrusive, use only what you need, supports advanced capabilities
- Capabilities include:
  - **Dependency Injection** (**Inversion of Control/IOC**) to manage bean dependencies
  - **DAO/Repository/Transaction** support for persistent data
  - **ORM** support (Object-relational mapping, e.g. JPA)
  - **AOP**: Aspect-oriented programming
  - **Web**: Integration with standard JEE Web apps
  - **Spring MVC**: Model-View-Controller Web framework
  - **Security**: Authentication and authorization

The Spring Modules



The Spring Distribution

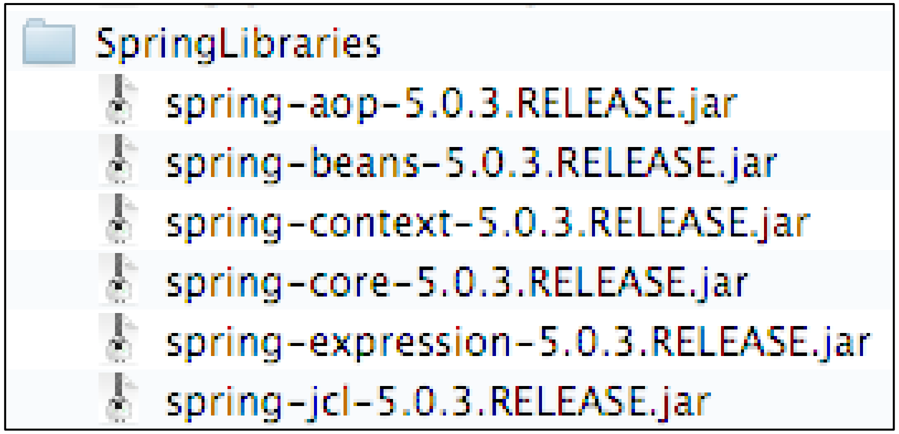
- Spring home page: **http://spring.io/**

- Distributed as modules in separate jars
  – e.g. *spring-beans-5.0.5.RELEASE.jar*
  – Has external dependencies - e.g. logging, JUnit, etc. (1)
  – Generally use a tool like **maven** for dependencies
    - We supply jars the jars for some labs, and use maven in others

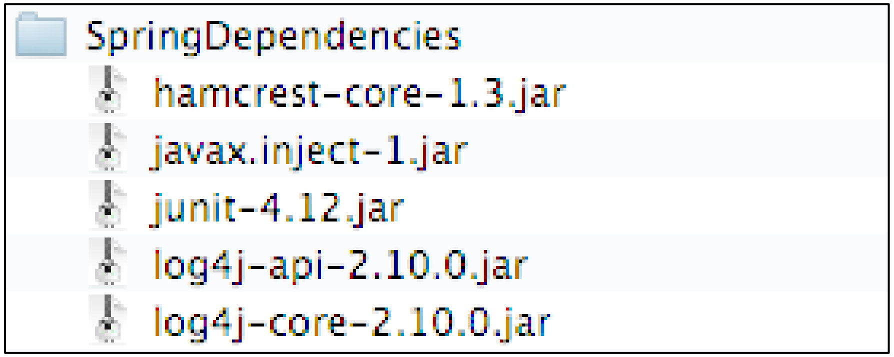
- Spring vs. JEE (Java Enterprise Edition) (2)
  – JEE similarly supports enterprise apps
    - e.g. CDI for lifecycle / dependency injection
  – Which to use? What works best
    - Based on your current and future system needs
  – You might use both - e.g. a JEE Web container
    - With Spring for lifecycle management and DI
    - Or Spring MVC layered on top of JEE Servlets/JSP

The Spring jars

- At right, are the Spring libraries we supply for the early labs (1)
  - They are a subset of Spring
  - Later labs, which need more jars, use maven for dependencies



- These are external dependencies (2)
  - Again, just a subset of what we'll need in later labs



A Word About JUnit

- **JUnit**: Open source Java testing framework
  - Often used in examples and labs to test our work
  - Labs also create console output - that's not standard (see notes)
- JUnit capabilities include:
  - **Annotations** for declaring test methods (e.g. **`@Test`**)
  - **Assertions** for testing expected results
  - **Test fixtures** for sharing common data
  - **Test runners** for running tests
- See next slide for an example
  - We'll review in more depth in the Testing session
- Most development environments have JUnit support
  - We'll use them to run tests which drive the lab code
  - The tests are the `@Test` annotated methods (see next slide)

JUnit Example

- To write a JUnit test, we:
  - Create a class, one or more methods annotated with **`@Test`**
  - Make **assertions** using static methods in **`org.junit.Assert`** (e.g. **`assertTrue`**)
- Note the **`@Test`** on `testContextNotNullPositive()`
  - The test creates the application context, and checks that it's non-null
  - We use `org.junit.Assert.`**`assertNotNull`** to perform the test
  - See notes about `import static` and `assertTrue` usage

```
// JUnit relevant code shown - some imports / code omitted
import static org.junit.Assert.*;
import org.junit.Test;

public class SpringTests {
   @Test
   public void testContextNotNullPositive() {
      ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext();
     // Just an example - we'd probably never test that the new operator works
      assertNotNull("spring container should not be null", ctx);
   }
}
```

Lab 1.1: Setting Up the Environment

In this lab you'll set up the lab environment, boot the Spring container, and test it with a unit test

Spring Introduction

Overview

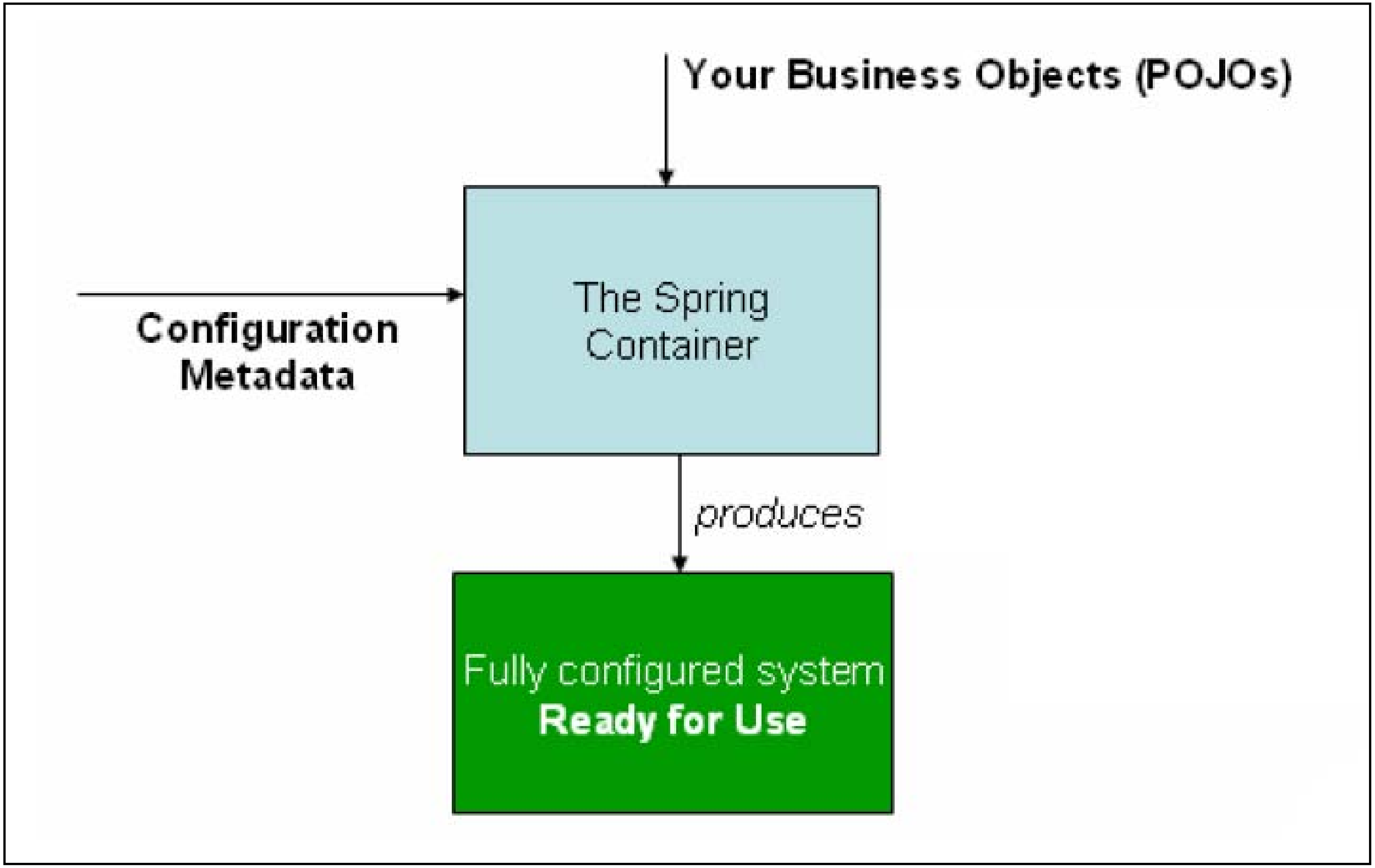
Spring Introduction

Dependency Injection

Managing Beans: Core Spring Capability

- **Beans**: Fancy name for application objects
  - They're **POJO**s (Plain Old Java Objects)
- The **Spring container** is the "manager"
  - Uses **configuration information** (metadata) to **define, instantiate, configure, and assemble** beans
    - **Metadata**: Information about your beans (e.g. bean definitions)
  - Container uses the configuration to create and manage beans (1)
- **Configuration choices** include XML and annotations
  - **XML**: The "classic" configuration from early Spring
  - Two **annotation** choices (**`@Component`**, **`@Configuration`**)
  - We'll cover all of these

A Basic Spring Application



The JavaTunes Online Store

- The course uses JavaTunes as an example and lab domain
  - A simple online music store [(1)]
- Some of the types you'll see include:
  - **`com.javatunes.domain.MusicItem`** : JavaBean-style value class representing a music item (e.g. an mp3)
  - **`com.javatunes.service.Catalog`** : Interface defining JavaTunes catalog functionality (including search)
  - **`com.javatunes.service.CatalogImpl`** : Concrete `Catalog` implementation (uses `ItemRepository`)
  - **`com.javatunes.persistence.ItemRepository`** : Interface defining data access API for items
  - **`com.javatunes.persistence.InMemoryItemRepository`** : Concrete `ItemRepository` implementation (simple in-memory storage)

Some JavaTunes Types

- `Catalog` and `CatalogImpl` are shown below
  - Note how `CatalogImpl` implements the `Catalog` interface (1)
- Let's look at how to configure some objects

```
package com.javatunes.service;

public interface Catalog {
  public MusicItem findById (long id);
  public Collection<MusicItem> findByKeyword(String keyword);
  public long size();
}
```

```
package com.javatunes.service;

public class CatalogImpl implements Catalog { // Detail omitted

  public MusicItem findById (long id) { /*  */ }
  public Collection<MusicItem> findByKeyword(String keyword)
    { /*  */ }
  public long size() { /*  */ }
}
```

XML Configuration Example

- Spring can be configured in an XML file
  - Default config file: ***applicationContext.xml***, but can be anything [1]
  - General structure: Top level **<beans>** containing **<bean>** elements
    - Each <bean> defines a bean
    - Generally specify **`id`** (a name) and **`class`** (fully qualified class name) [2]
  - Supports other configuration also - we'll cover it as we encounter it
- At bottom, we define one bean with XML (the metadata)
- Many existing applications use XML
  - But annotation-based approaches now recommended

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- The beans namespace is the default one for the document -->
<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="musicCatalog" class="com.javatunes.service.CatalogImpl"/>

</beans>
```

The Spring Container

- The Spring container
  - Reads your configuration, and based on it:
    - **Instantiates/initializes** application objects
    - **Resolves object dependencies**

- `org.springframework.context.`**`ApplicationContext`**
  - Core API to **access the Spring container** in your code
  - Several implementations with different capabilities, e.g.
  - **`ClassPathXmlApplicationContext`**: Loads XML resources from the class path

- Interface `ApplicationContext` extends **`BeanFactory`**
  - `BeanFactory` has many core methods - usually not used directly

Instantiating and Using the Container

- Below, we create a **`ClassPathXmlApplicationContext`**
  – Sometimes called "instantiating the container"
  – We pass the name of our config file (or multiple file names)
  – The container will create and manage the beans defined in the XML
- Next, we look up our configured bean via the context
  – Looked up **by type** (`Catalog.class`) - can look up **by name** also (1)
  – We do any needed work, then close the context

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

  // Code fragment - other imports and much detail omitted
  ClassPathXmlApplicationContext ctx=
    new ClassPathXmlApplicationContext("applicationContext.xml");

  // Note that getBean uses Java generics (no casting needed)
  Catalog cat = ctx.getBean(Catalog.class);
  // Or lookup by name: ctx.getBean("musicCatalog", Catalog.class);

  MusicItem item = cat.findById(1L); // Use our bean
  ctx.close(); // Close the context
```

Why Bother - What do we Gain?

- Main benefit: Our code **doesn't know about `CatalogImpl`**
  - It just knows about needed functionality (interface **`Catalog`**)
  - We've **decoupled** our code **from a dependency on the implementation** class
- Can use **any implementation** in our configuration
  - Client code **will not change**
  - That's why we **code to interfaces**, not concrete types
- Bean lifecycle is **managed by container**
  - We don't instantiate our beans directly
  - Very useful for more complex systems
  - We'll see more useful capabilities soon

Summary: Working With Spring

- **Create Spring configuration data** for your beans
  - It's the "cookbook" telling Spring how to create objects
  - Via an XML file like *applicationContext.xml* or via annotations

- **Initialize the Spring container**
  - e.g. **create an application context** instance to read config data
  - It will initialize the beans in the config file(s)

- **Retrieve beans** via the context instance and use them
  - e.g. use **`getBean()`** to look up a bean by type or name
  - **Lookup by type is preferred**, unless you can't do it
    - For instance, if you have two beans implementing the same type

More on ApplicationContext

- Access point for many Spring capabilities, including:
  - **Bean access**
  - **Resource Access**: Config files, URLs, other files
  - **Message resources with I18N** capability
  - **Publishing events** to managed beans that are listeners
  - **Multiple, hierarchical contexts**
- **`ClassPathXmlApplicationContext`**
  - Loads XML config files from the classpath
- **`FileSystemXmlApplicationContext`**
  - Loads XML config files from the file system or URLs
  - Both in `org.springframework.context.support`
- **`AnnotationConfigApplicationContext`**
  - Accepts annotated classes as input (more on this later)
  - In `org.springframework.context.annotation`

Some BeanFactory/ApplicationContext API

- Useful methods include:
  - `boolean containsBean(String)`: true if named bean exists
  - `<T> T getBean(Class<T> requiredType)`: Get by type
  - `<T> T getBean(String, Class<T> requiredType)`: Get by name
  - `Class<?> getType(String name)`: Get type of named bean
  - `boolean isSingleton(String)`: Is bean a singleton
  - `String[] getAliases(String)`: Get any aliases for this bean
  - Many more methods - see the javadoc

- Can specify config files in multiple ways
  - ant-style wildcards: e.g. ***conf/**/ctx.xml*** - All *ctx.xml* files under *conf*
  - **`file:`** and **`classpath:`** prefixes - forces use of specified location, e.g.
    - The following loads from the classpath:
    - `new FileSystemXmlApplicationContext("classpath:ctx.xml");`
  - Spring uses its resource classes under the hood to do this

Mini-Lab: Review Javadoc

Mini-Lab

- We provide the Spring Javadoc
  - Under *StudentWork/Spring/**Resources/SpringDocs/javadoc***
- In a browser, open *index.html* in the folder above
- Review the javadoc for the following types
  - `BeanFactory`
  - `ApplicationContext`
  - `ClassPathXmlApplicationContext` (especially the constructors)
  - `FileSystemXmlApplicationContext` (especially the constructors)

Lab 1.2: Hello Spring World

In this lab, we will create and use a Spring context
to access a bean instance

Annotation-Based Configuration Example

- Beans can be declared with annotations
  - **@Component** (`org.springframework.stereotype`) - Spring specific
    - `@Component` often called a "stereotype" annotation
  - **@Named** (`javax.inject`) - Standard Java (JSR 330) annotation
  - We'll use Spring style annotations in this course [1]

- Below, **@Component** declares `CatalogImpl` as a bean
  - Specifies id as **`musicCatalog`** (Default id: `catalogImpl`)
    - Same bean as previous XML example
  - Could also have used **`@Named("musicCatalog")`**

```
import org.springframework.stereotype.Component;

package com.javatunes.service;

@Component("musicCatalog") // Declares bean with id musicCatalog
public class CatalogImpl implements Catalog {
 /* Most detail omitted … */
}
```

A Brief Note on Annotations

- Standard Java mechanism to **add metadata** to source code
  - Like comments that the compiler is aware of
- **@** is used for both declaration and use of annotations (1)
  - e.g. **`@Component`**
- Annotations don't directly affect program semantics
  - They are **not executable code**
- Tools work with the annotated code
  - And may affect the semantics of a running program
  - **`@Named`**/**`@Component`** annotated beans are recognized as bean defs by the Spring container

Enabling Annotations / Detecting Beans

- Spring can automatically **scan** for annotated beans on the classpath
  - This registers them as normal beans

- Enable auto scanning via `<context:component-scan/>`
  - A standard element from Spring's `context` namespace (see next slide)
  - Includes the capabilities of `<context:annotation-config/>` (1)
  - **basePackage** attribute configures the packages to scan

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

  <context:component-scan base-package="com.javatunes"/>

</beans>
```

Overview - Spring's XML Schemas

- Spring provides XML Schemas for configuration
  - With custom namespaces and tags with complex behavior [1]
  - e.g. the **`context:`** namespace we just used
- Spring namespaces include:
  - **`aop`**: Configures AOP support
  - **`beans`**: The standard bean tags we've seen already
  - **`context`**: Configures ApplicationContext related things
  - **`jee`**: JEE related configuration such as looking up a JNDI object
  - **`jms`**: Configures JMS related beans
  - **`lang`**: Exposing objects written in language like JRuby as beans
  - **`tool`**: Adds tooling-specific metadata
  - **`tx`**: Configures transaction support
  - **`util`**: Common, utility configuration issues

Lab 1.3: Spring Annotations

In this lab, we'll work with Spring Annotations

Dependency Injection

Overview

Spring Introduction

Dependency Injection

Dependencies Between Objects

- In OO systems, multiple objects work together
  - e.g., Object A directly uses Object B to accomplish a goal (1)
  - So Object A **depends on** Object B
- Direct dependencies have several issues
  - **Rigid**: Changes affect other areas, so are hard to make
  - **Fragile**: Changes cause unexpected failures in other areas
  - **Immobile**: Hard to reuse functionality
    - Modules can't be disentangled
- We'll illustrate a direct dependency
  - Then show an alternative approach

Example of a Direct Dependency

- `CatalogImpl` uses `InMemoryRepository`
  - And creates an `InMemoryRepository` instance directly
  - `CatalogImpl` **depends** on the lower level module details
  - To use a different data store - e.g. a `FileRepository`, `CatalogImpl` must change (see notes)

```
public class InMemoryItemRepository {
  public MusicItem get(Long id) { /* Details not shown */  }
}
```

```
public class CatalogImpl implements Catalog {
  InMemoryItemRepository rep = new InMemoryItemRepository();
  public MusicItem findById(long ID) {
    return rep.get(id);
  }
}
```

Dependency Inversion

- All modules **depend on abstractions**, not each other
  - In other words - **program to interfaces**
  - `CatalogImpl` only knows about the abstract `ItemRepository`
    - Can be initialized with another type (e.g. `FileItemRepository`)
    - `CatalogImpl` **need not change** - the modules are **decoupled**

```
public interface ItemRepository {
   public MusicItem get(Long id);
}
```

```
// Much detail omitted …
public class InMemoryItemRepository implements ItemRepository { /*…*/ }
```

```
public class CatalogImpl implements Catalog {
   private ItemRepository itemRepository; // get/set methods not shown
   public MusicItem findById(Long id) { return itemRepository.get(id); }
}
```

```
  // Code fragment - most detail omitted
  InMemoryItemRepository rep=new InMemoryItemRepository();
  CatalogImpl catalogImpl=new CatalogImpl();
  catalogImpl.setItemRepository(rep);
  MusicItem found = catalogImpl.findById(1);
```

Dependency Injection (DI) in Spring

- The Spring container **injects** dependencies into a bean
  - Into bean properties, constructors, or via factory method args
- Dependencies **are defined in the Spring configuration**
  - Spring initializes the dependencies ("injects" them) based on the config
  - **No need to explicitly initialize dependencies** in your code
- We illustrate XML config of this below (injecting via a **set method**)
  - **`<property …>` injects** into the catalog's `itemRepository` property
  - Automatically done when the container creates the bean

```
<beans … > <!-- Much detail / Namespace declarations omitted -->

   <bean id="inMemoryRepository"
      class="com.javatunes.persistence.InMemoryItemRepository"/>

   <bean id="musicCatalog" class="com.javatunes.service.CatalogImpl">
     <property name="itemRepository" ref="inMemoryRepository"/>
  </bean>

</beans>
```

Injection with Autowired

- Can also inject with **@Autowired** - shown below for the repository
  - This injects a dependency **by type** - same result as previous XML
- At bottom, we get a catalog from Spring
  - With an already initialized repository (by Spring's DI)
  - We're ready to work !

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

@Component("musicCatalog") // Declares bean - most detail omitted
public class CatalogImpl implements Catalog {
  @Autowired  // can also apply to setter method or constructor
  private ItemRepository itemRepository;
}
```

```
public class CatalogTests {  // imports, other detail omitted
  @Test testCatalogFindById() {
    ClassPathXmlApplicationContext ctx =
      new ClassPathXmlApplicationContext("applicationContext.xml");
    Catalog cat = ctx.getBean(Catalog.class);  // See note (1)
    assertNotNull("item shouldn't be null", cat.findById(1L)); // Use cat
}
```

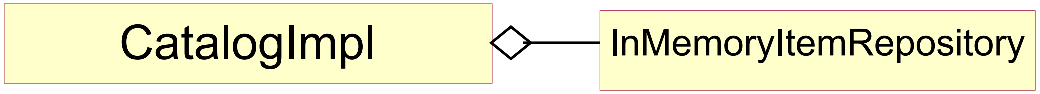
@Named/@Inject (JSR-330) Example

- Below, **@Named** is used to declare a bean
  - **`@Inject`** is used to inject the dependency
- Results are the same as with `@Component` style
  - And the test client would look exactly the same
- We'll use **@Component** style going forward, as mentioned earlier
  - It's the more common choice

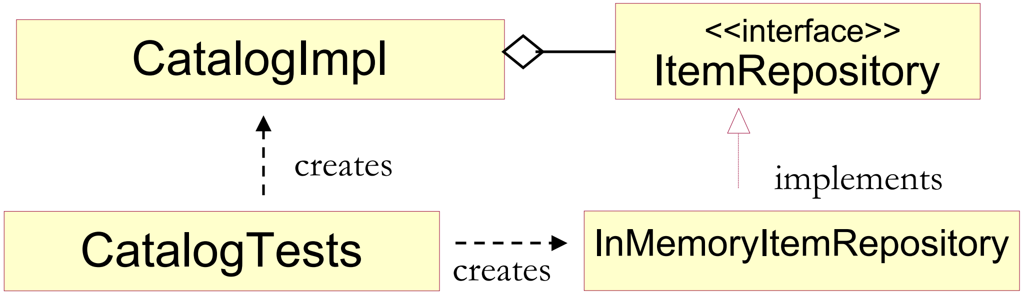
```
import javax.inject.Inject;
import javax.inject.Named;

@Named("musicCatalog") // Declares bean - most detail omitted
public class CatalogImpl implements Catalog {
  @Inject
  private ItemRepository itemRepository;
}
```

Dependency Injection Reduces Coupling

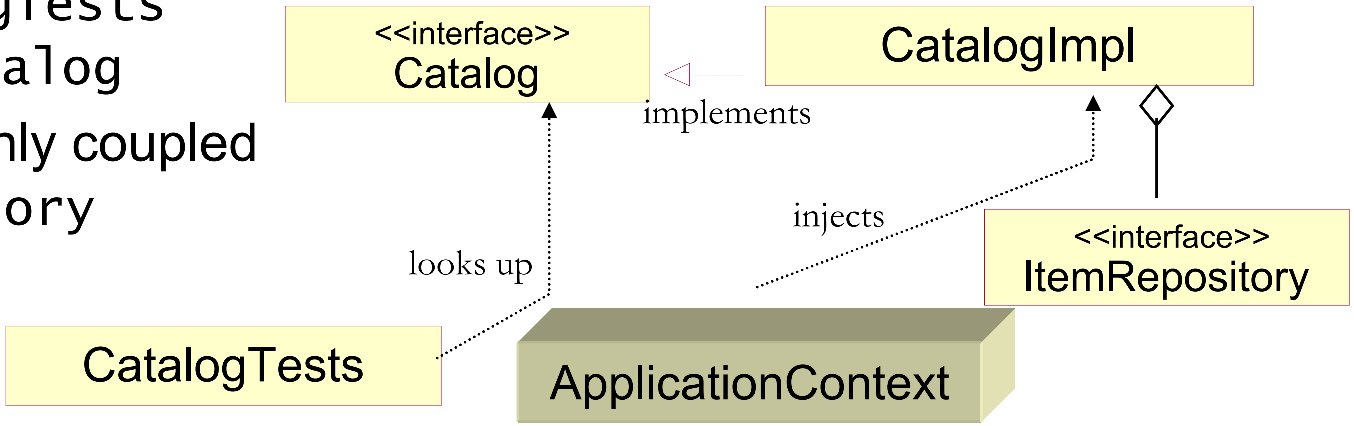
- The simplest case, `CatalogImpl` coupled to `InMemoryItemRepository`



- `CatalogImpl` coupled to `ItemRepository` only (Dependency Inversion)
  - `CatalogTests` coupled to `CatalogImpl` and `InMemoryItemRepository`



- Using DI – `CatalogTests` only coupled to `Catalog`
  - `CatalogImpl` only coupled to `ItemRepository`



Advantages of Dependency Injection

- **Hides dependencies** and **reduces coupling in your code**
  - Coupling is basically a measure of the dependencies
- We see this in two ways:
  - `CatalogImpl` is not coupled to `InMemoryItemRepository`
  - `CatalogTests` is not coupled to `CatalogImpl` or `InMemoryItemRepository`
- The dependencies are still there but **not in the code**
  - Dependencies are **moved to the spring configuration**
  - They're injected into beans without you coding it
  - Commonly referred to as **wiring** beans together
- This leads to **more flexible** code that is **easier to maintain**
  - At a cost – using Spring, and maintaining the spring configuration

Constructor Injection

- Can easily inject into a constructor (ctor)
  - Assume the `CatalogImpl` constructor shown below
- With XML, **`<constructor-arg>`** means "inject into constructor"
- With **`@Autowired`**, apply it to the constructor
- See notes for some additional detail/capabilities

```
public class CatalogImpl implements Catalog { // Most detail omitted
  public CatalogImpl(ItemRepository repository) { /* … */ }
}
```

```
  <!-- XML config - Other detail omitted -->
  <bean id="musicCatalog"
        class="com.javatunes.service.CatalogImpl">
    <constructor-arg ref="itemRepository"/> <!-- inject via ctor -->
  </bean>
```

```
public class CatalogImpl implements Catalog { // Most detail omitted
  @Autowired // Inject into ctor below
  public CatalogImpl(ItemRepository repository) { /* … */
}
```

Setter Injection vs. Constructor Injection

- **Setter Injection Pros**
  - **Easy**, **Flexible**: Simple setters, easily choose properties to configure
  - Good for **optional properties** (with defaults to reduce not-null checks)
  - Supports **reconfiguration** after startup
- **Setter Injection Cons**
  - **Doesn't guarantee property initialization** (Forget to do it - BUG!)
  - **Setter methods are required**

- **Constructor Injection Pros**
  - **Guaranteed Initialization**: Immediate error if you leave it out (1)
  - Good for **required properties** (can be **immutable** if setter omitted)
- **Constructor Injection Cons**
  - **Unwieldy and Brittle**: Multiple constructors with multiple args unwieldy
    - Refactor if you have this
  - **Less flexible**: Need ctors for each scenario, and can't reset properties

Qualifying Injection by Name

- Consider two repository implementations (first two examples below)
  - What happens if we try to auto-wire one in?
  - The first autowire below **fails at runtime** - which one to inject?
  - The second try is "**qualified**" by the bean name using **`@Qualifier`**
  - It injects based on the bean name

```
@Component // Declare as bean - default id inMemoryItemRepository
public class InMemoryItemRepository implements ItemRepository {…}
```

```
@Component // Declare as bean - default id cloudItemRepository
public class CloudItemRepository implements ItemRepository {…}
```

```
@Component public class CatalogImpl implements Catalog { // Autowire
  @Autowired private ItemRepository itemRepository; // FAILS!
}
```

```
import org.springframework.beans.factory.annotation.Qualifier;

@Component public class CatalogImpl implements Catalog { // Autowire
  @Autowired @Qualifier("cloudItemRepository")
  private ItemRepository itemRepository; // Injects the Cloud-based
}
```

Lab 1.4: Dependency Injection

In this lab, we'll work with Spring's DI capabilities

Review Questions

- What is Spring, and how does it help you build enterprise apps?
- What is **Dependency Inversion**?
- What is **Dependency Injection**, and how does it work in Spring
- What is an `ApplicationContext`?
- How does Spring use **annotations** on your bean classes for defining and wiring beans?
  - What are the pros/cons of using them?

Lesson Summary

- **Spring**: Lightweight enterprise framework that supports:
  - Dependency Injection
  - Persistence support (Repository/DAO and ORM)
  - Integration with standard Web technologies, and MVC Web apps
- **Manages complex dependencies** - non-intrusive and lightweight
  - Supports **loose coupling** between components
  - Encourages **coding to interfaces** (good design)
    - Also called **Dependency Inversion**
- **Uses configuration metadata** to initialize beans and dependencies
  - Via XML configuration files or annotations
- **Dependency Injection**: Injects dependencies based on config
  - Complete **decoupling** from concrete implementation types

Lesson Summary

- **`ApplicationContext`**: API to the Spring container functionality
  - Configure/wire beans, access program resources, work with resource bundles, load multiple contexts, and publish events to beans
  - Common implementations include:

  `ClassPathXmlApplicationContext`,

  `FileSystemXmlApplicationContext`, and

  `AnnotationConfigApplicationContext`

- Provides resource access in a flexible and powerful way
  - From file system, the classpath, URL access, and more
  - Supports ant-style wildcards like ***conf/**/ctx.xml***
  - Uses its own **resource classes** to accomplish resource access

Session 2: Configuration in Depth

Java-based Configuration
Integrating Configuration Types
Bean Scope and Lifecycle
Externalizing Properties
Profiles

Lesson Objectives

- Learn how to use **`@Configuration`** (**Java**-based) config style

- **Integrate** the various styles of configuration
  - Use XML from `@Configuration`
  - Use `@Configuration` from XML
  - Use `@Component` when convenient and appropriate

- Understand the **lifecycle and scope** of managed beans
  - Configure singleton and prototype beans
  - Work with Spring bean lifecycle events

Java-based Configuration

Java-based Configuration
Integrating Configuration Types
Bean Scope and Lifecycle
Externalizing Properties
Profiles

Java Configuration Overview

- **Java-based** Spring configuration (added in Spring 3)
  - **@Configuration** annotated classes contain config info
    - **Decouple configuration** from source code (like XML)
  - **@Bean** annotated methods define a bean
- Below, we configure one bean (same as XML at bottom)

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SpringConfig {
  @Bean
   public ItemRepository itemRepo() {
      return new InMemoryItemRepository();
   }
}
```

```
<bean id="itemRepo"
   class="com.javatunes.persistence.InMemoryItemRepository"/>
```

Using Java-based Configuration

- **AnnotationConfigApplicationContext** recognizes `@Configuration` classes [1]
  - Just pass in the config class (or comma separated classes)
  - Bean lookup remains the same

```
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

// much detail omitted in this example
public class ConfigurationTests {
  @Test
  public void testRepoLookupNotNull() {

    AnnotationConfigApplicationContext ctx =
      new AnnotationConfigApplicationContext(SpringConfig.class);

    ItemRepository rep = ctx.getBean(ItemRepository.class);
    assertNotNull("repository shouldn't be null", rep);
    ctx.close();
   }
}
```

Dependency Injection

- Wire a bean by referencing its @Bean method
  - Below we inject `itemRepository` into `musicCatalog`
  - Same as XML at bottom

```
@Configuration
public class SpringConfig {
  @Bean
   public ItemRepository itemRepo() {
     return new InMemoryItemRepository();
   }
  @Bean
  public Catalog musicCatalog() {
    CatalogImpl ci = new CatalogImpl();
    ci.setItemRepository(itemRepo());
    return ci;
  }
}  // This is equivalent to the XML config below
```

```
<bean id="itemRepo“
   class="com.javatunes.persistence.InMemoryItemRepository"/>
<bean id="musicCatalog" class="com.javatunes.service.CatalogImpl">
  <property name="itemRepository" ref="itemRepo"/>
</bean>
```

How Does it Work ?

- An `@Configuration` class **is metadata**
  - Like an XML config file – just a different format
- `@Bean` methods **not called like normal methods**
  - Interpreted as metadata indicating a bean definition
  - **ONLY** run by the container - and completely under its control
  - "Calling" the method in a config class **specifies an injection**
- Note: The code below **uses the same ItemRepository instance** as on the previous slide (by default a bean is a singleton)
  - Looks confusing, but remember – this is **metadata**, not normal code (1)

```
// Assume rest of @Configuration class same as on previous slide
@Bean
public Catalog anotherCatalog() {
  CatalogImpl ci = new CatalogImpl();
  ci.setItemRepository(itemRepo());
  return ci;
}
```

Dependencies in Configuration Classes

- You can inject dependencies into `@Configuration` classes
  - Flexible - can inject beans configured elsewhere (e.g. in XML)
  - Allows organizing configuration to meet your needs

```
@Configuration
public class ServiceConfig {
  @Autowired
  private ItemRepository repo;

  @Bean
   public Catalog musicCatalog(){
     CatalogImpl ci = new CatalogImpl();
     ci.setItemRepository(repo);
     return ci;
   }
}
```

```
@Configuration
public class RepositoryConfig {
  @Bean
  public ItemRepository itemRepo() { return new InMemoryItemRepository(); }
}
```

Injecting Configuration Classes

- `@Configuration` classes can inject another config class
  - To explicitly navigate from one config class to another
  - Useful when integrating multiple configuration classes

```
@Configuration
public class SpringConfig {
  @Autowired
  private RepositoryConfig repoConfig;

  @Bean
   public Catalog musicCatalog(){
     CatalogImpl ci = new CatalogImpl();
     ci.setItemRepository(repoConfig.itemRepo());
     return ci;
   }
}
```

```
@Configuration
public class RepositoryConfig {
  @Bean
   public ItemRepository itemRepo() {
     return new InMemoryItemRepository();
   }
}
```

Other @Bean Capabiliites

- A bean can specify a name

```
@Bean(name="musicCatalog") // Bean is named musicCatalog
public Catalog getCatalog() { /* … */ }
```

- You can have multiple aliases using a string array

```
@Bean(name={ "musicCatalog", "albumList" })
```

  – The example uses the standard Java array initialization syntax

- You can specify init / destroy methods

```
@Bean(initMethod="init", destroyMethod="cleanup")
```

  – Called at appropriate times when instance created

- More capabilities that we'll cover as we use them
  – @Bean javadoc is useful for details
  – As is the Spring reference manual

Mini-Lab: Review Javadoc

Mini-Lab

- Open Javadoc in browser if not open (see Session 1 Mini-Lab)
- Review the javadoc for the following types
  - `@Configuration`
  - `@Bean`
  - `AnnotationConfigApplicationContext` (especially the constructors)

Integrating Configuration Types

Java-based Configuration
Integrating Configuration Types
Bean Scope and Lifecycle
Externalizing Properties
Profiles

XML and @Component Pro / Con

- **XML pros**:
  - Separates configuration from Java source - no recompile on changes
- **XML cons**:
  - String/name matching is **error prone**, not caught until runtime, fragile
  - **Verbose** and yet another point of failure
  - Have to keep XML config and Java source in sync
  - Can't see wiring points by looking at Java Source
- **@Component/@Autowired pros**:
  - **Simple and easy** for some situations (so a reasonable choice then) (1)
  - Configuration is close to Java source
- **@Component/@Autowired cons**:
  - Brittle - may not be adequate for non-trivial object models
  - **Only one bean instance per type** can be configured (2)
  - Configuration scattered across many files
  - Components less reusable - annotations per type, not per instance

Java-based Configuration: Pro / Con

- Pros:
  - **No XML**, Java used for configuration, fairly compact and readable
  - Configuration is **external to bean code**
  - **Type safe** and **no string matching**
    - The compiler checks most things
    - In XML, there are a lot of strings and many errors, like misspelling a classname, may not be caught until runtime
  - Full control over **bean definition** (e.g. two beans of same type is easy)
  - **Refactoring friendly** – uses normal Java tools
  - **Mix and match** with XML configuration (e.g. existing XML config or DB connection info that is easier to do in XML)
- Cons:
  - Configuration changes require **recompilation**
  - Somewhat **obscure syntax** - you really need to get used to it
  - **Verbose** in some areas

Choosing a Configuration Style

- We've seen three configuration styles (XML, Java, annotation)
  – All are useful, so which do you use? (1)
- **XML**: Complete configuration abilities, provides high-level tags (2)
  – However - it's XML, with the issues we've discussed
  – May still be useful for real configuration info (e.g. DB connection info)
  – You may have older code that uses it (and you can keep it)
  – Or you may like it - so use it !
- **`@Component/@Named`**: Simple/Fast - but support only simple config
  – Use for **simple** definitions (and wiring with auto-wiring)
  – Not a complete solution
- **`@Configuration`**: Fairly complete, and supports complex bean configurations
  – Use for **more complex** bean configurations
  – Good choice for **new projects**

Integrating Configuration Metadata

- Two main scenarios
  - Integrating similar metadata defined in different places
    - e.g. integrating multiple `@Configuration` classes
  - Integrating metadata of different types
    - e.g. definitions using all three types of metadata
    - XML, `@Component`, and `@Configuration`

- We'll take a brief look here
  - We'll use various techniques in the labs as appropriate

@Import: @Configuration by @Configuration

- **@Import** an `@Configuration` class into another
  - Below, `SpringConfig` imports `ServiceConfig`, `RepositoryConfig`
  - Supports simple top-level config, and separation of responsibilities [1]

```
// Details not shown
@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SpringConfig { /* May be empty */ }
```

```
@Configuration
public class ServiceConfig {
  @Autowired
  private RepositoryConfig repConfig;
  @Bean
   public Catalog musicCatalog(){ /* Detail omitted */ }
}
```

```
@Configuration
public class RepositoryConfig {
  @Bean
   public ItemRepository itemRepository() { /* Detail omitted */ }
}
```

<import>: XML by XML

- `<import>` one XML file into another
  - Below, we assume *service.xml* and *repository.xml* are config files
- Supports good organization - as with `@Import`
  - Simple top-level config, and separation of responsibilities

```
<beans … > <!-- applicationContext.xml - Namespace declarations omitted -->
  <import resource="service.xml"/>
  <import resource="repository.xml"/> <!-- Not shown -->
</beans>
```

```
<beans … > <!-- repository.xml - Namespace declarations omitted -->
   <bean id="inMemoryRepository"
      class="com.javatunes.persistence.InMemoryItemRepository"/>
  </bean>
</beans>
```

```
<beans … > <!-- service.xml - Namespace declarations omitted -->
  <bean id="musicCatalog" class="com.javatunes.service.CatalogImpl">
     <property name="itemRepository" ref="inMemoryRepository"/>
  </bean>
</beans>
```

Importing Between XML/@Configuration

- `@Configuration` can import XML
  - For example, to import a file *otherConfig.xml* on the classpath

```
@Configuration
@ImportResource("classpath:otherConfig.xml")
public class SpringConfig { … }
```

- XML can include `@Configuration` - just use **<bean>**
  - `@Configuration` classes are also `@Components` (1)
    - They will still be treated as metadata
  - Below, we include the beans in `RepositoryConfig`

```
<beans … > <!-- Namespaces omitted -->

  <bean class="com.javatunes.config.RepositoryConfig"/>

</beans>
```

Scanning for @Configuration Classes

- `@Configuration` classes can be auto-detected
- **`@ComponentScan`** configures scanning with config classes
  - Must still bootstrap at least one class (1)
  - Other `@Configuration` classes can be picked up by scanning
- With XML, can scan with `<context:component-scan/>`
  - Since `@Configuration` classes are also `@Component` classes

```
@Configuration
// Scan com.javatunes and subpackages
@ComponentScan(basePackages = "com.javatunes")
public class SpringConfig  {
    ...
}
```

Lab Options

- The next lab will demonstrate the use of Java-based configuration
- In future labs, we'll generally use Java-based configuration
  - Some labs may have some other configuration
  - For example, DB/datasource configuration could use XML
    - Clearer, and lets you change DB connection info without recompiling
    - Can also use properties files + Java configuration with same result
- Some labs are supplied with **alternate XML-based** versions
  - Still a lot of code (legacy and current development) using XML
  - We'll note that in the lab when we do it

Lab 2.1: Java-based Configuration

In this lab, we will use `@Configuration` classes to configure all our beans and dependencies

Bean Scope and Lifecycle

Java-based Configuration
Integrating Configuration Types
Bean Scope and Lifecycle
Externalizing Properties
Profiles

Bean Scope

- Spring beans are **singletons** by default
  - **One instance** (only) created by container
  - **Used for all access** to that bean, including via DI and `getBean()`

- Other scopes include:
  - **`singleton`** (the default): Bean definition produces a **single object instance** per Spring container
    - Any request for that bean returns that instance
  - **`prototype`**: Bean definition produces **multiple object instances**
    - **New instance created** every time a bean is needed
    - Useful when instances **have state**
  - **request, session, global session** : Only valid in the context of a web-aware `ApplicationContext`, and not covered now

Specify Bean Scope - XML

- Below we configure a caching bean for an application (1)
  - Assume it caches data for repositories (It has state!)
  - Scoped as **prototype**
    - Instance created every time the cache bean is used
  - **Two instances** created by the config below
    - **Two references** - one in each repository

```
<!-- Much detail omitted -->
<bean id="cache" class="com.javatunes.persistence.CacheImpl"
       scope="prototype">
</bean>

<bean id="localRepository" class="com.javatunes.persistence.JPAItemRepository"
    <property name="cache" ref="cache"/>
</bean>

<bean id="cloudRepository" class="com.javatunes.persistence.CloudItemRepository"
    <property name="cache" ref="cache"/>
</bean>
```

Specify Bean Scope - @Scope

- Spring's **@Scope** specifies a bean scope
  - Via `ConfigurableBeanFactory` constants
  - e.g. a singleton:
    `@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)`

- In `@Component` classes annotate the class (below)

- In `@Configuration` classes annotate `@Bean` methods (bottom)

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class CacheImpl implements Cache { /* ... */ }
```

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public Cache cache() {
  return new CacheImpl();
}
```

Bean Creation/Destruction Lifecycle

- Programs can interact with a bean's lifecycle
  - Annotate methods with JSR-250 annotations or use XML
  - Requires `<context:annotation-config/>` processors (1)
  - Below, `init()` is called after creation, `destroy()` before destruction
- The complete lifecycle is complex (2) - we'll cover it as needed

```
// Much detail omitted …
import javax.annotation.*;
public class InMemoryItemRepository implements ItemRepository {
  @PostConstruct
  public void init() { /* … */ }
  @PreDestroy
  public void cleanup() { /* … */ }
}
```

```
  <!-- Much detail omitted -->
  <bean id="inMemoryRepository"
        class="com.javatunes.persistence.InMemoryItemRepository"
        init-method="init" destroy-method="cleanup"   />
```

[Optional] Bean Creation Lifecycle Details

- Bean creation is quite complex in Spring
- Spring has interfaces to interact with all parts of this lifecycle *
  - **Instantiation**: Container creates the bean instance
  - **Populate properties**: Container injects bean's properties
  - **Set bean name**: If bean implements `BeanNameAware`, call `setBeanName()` passing in bean's name
  - **Set bean factory**: If bean implements `BeanFactoryAware`, call `setBeanFactory()` passing in the bean factory
  - **Set application context**: If bean implements `ApplicationContextAware`, call `setApplicationContext()`
  - **Postprocess** (before init): If there are any `BeanPostProcessors`, call their **`postProcessBeforeInitialization`**`()` methods
  - **Initialize bean**: If bean implements `InitializingBean` , call `afterPropertiesSet()`. If bean has a custom init method, call it
  - **Postprocess** (after init): If there are any `BeanPostProcessors`, call their **`postProcessAfterInitialization`**`()` methods

[Optional] BeanPostProcessor

- You can define post processors affecting every bean defined
  - Generally not needed with normal programming
- **First**: Define a class implementing **`BeanPostProcessor`**
  - This interface defines two methods:

```
Object postProcessBeforeInitialization(
                        Object bean, String beanName)
Object postProcessAfterInitialization(
                        Object bean, String beanName)
```

- **Second**: Define a bean using this class (1)
  - The container will call these methods for **every** bean it creates
  - At the two Postprocess points in the lifecycle shown earlier
  - You can define as many `BeanPostProcessors` as you want

[Optional] Event Handling

- `ApplicationContext` can notify beans of application events
  - Beans receive events if they implement `ApplicationListener`
  - The interface is parameterized, so only appropriately typed events are received (`ApplicationEvent` in the example below)
    - `ApplicationEvent` is the root of Spring's event hierarchy (1)
- The container itself publishes events, and user beans can publish custom events (2)

```
public interface ApplicationListener<E extends ApplicationEvent> {
  void onApplicationEvent(E event); // Handle an event
}
```

```
public class MyListener
                   implements ApplicationListener<ApplicationEvent> {
  public void onApplicationEvent(ApplicationEvent event) {
    System.out.println(event);
  }
}
```

Lab 2.2: Bean Lifecycle

In this lab, we will work with the bean lifecycle

Externalizing Properties

Java-based Configuration
Integrating Configuration Types
Bean Scope and Lifecycle
Externalizing Properties
Profiles

Externalizing Values in Properties Files

- Spring supports properties files to configure values [1]
  - Useful for data like DB properties (keeps it separate from other config)
  - Can modify props without recompile
  - Accessible via **placeholders** of form **`${propName}`**
- For `@Configuration` specify sources via **`@PropertySource`** [2]
  - **`PropertySourcesPlaceholderConfigurer`** activates placeholders
    - Must be declared as a static bean in a configuration class [3]
- XML config uses **`<context:property-placeholder … >`**
  - Reads the source and activates placeholders

```
<context:property-placeholder location="classpath:javatunes.properties"/> (4)
```

```
@Configuration
@PropertySource("classpath:javatunes.properties")
public class SpringConfig {
  @Bean
  public static PropertySourcesPlaceholderConfigurer propertyConfigurer() {
    return new PropertySourcesPlaceholderConfigurer();
  }
}
```

Accessing Externalized Properties

- **`@Value("${propName}")`** will inject a property value
  - We illustrate usage in `@Configuration` (below) and XML (bottom)
- Optionally provide default value like this

```
@Value("${jdbc.url:someDefaultValue}")
```

```
# javatunes.properties - only one property shown
jdbc.url=jdbc:derby://localhost:1527/JavaTunesDB
```

```
import org.springframework.beans.factory.annotation.Value;

@Configuration
public class RepositoryConfig {

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

}
```

```
<bean id="myDataSource" class="…">
  <property name="url" value="${jdbc.url}"/>
</bean>
```

The Spring Environment

- Externalized values also accessible in Spring environment
  – Easily autowired, as shown below
  – For programmatic access to properties

```
import org.springframework.core.env.Environment; // Detail omitted

@Configuration
public class RepositoryConfig {

  @Autowired
  Environment env;

  @Bean
  public Datasource myDataSource() {
    String url = env.getProperty("jdbc.url", String.class);
    // Other detail omitted
  }
}
```

SpEL: Spring Expression Language Overview

- Queries and manipulates object graphs
  - Can access beans, call methods, evaluate expressions …
- SpEL can be used in bean definitions
  - Expressions have the form **#{expression-string}**
  - Shown below using the pre-defined **systemProperties** bean
- Can use programmatically via **Expression**/**ExpressionParser**
  - Beyond scope of course - see Spring reference

```
// Code fragment from @Configuration class
@Value("#{systemproperties.jdbcUrl}")
String url;
```

```
<bean id="myDataSource" class="…">
  <property name="url" value="#{systemproperties.jdbcUrl}"/>
</bean>
```

Additional SpEL Examples

- Access a bean property

```
#{musicCatalog.size}
```

- Invoke method on bean (assume it has `clearHistory()` method)

```
#{musicCatalog.clearHistory()}
```

- Access static members of a class using the T (type) operator
  - You can call `java.lang.Math.random()` as follows

```
#{ T(java.lang.Math).random() }
```

- Can also be used to evaluate expressions in Java programs through the use of `ExpressionParser` and other classes
- See the reference manual for the wide range of SpEL's capabilities

Mini-Lab: Review SpEL Reference

Mini-Lab

- We provide the Spring Reference under
  - Under *StudentWork/Spring/**Resources/SpringDocs/reference***
- In a browser, open *index.html* in the folder above
  - Click on the link for the "Core" reference documentation
  - In the left hand column, click the link for the **Spring Expression Language (SpEL)**
    - Currently **Section 4** in the Core docs
- Spend 5 minutes reviewing the SpEL docs
  - Especially the subsection on "**Expressions in bean definitions**"
    - Currently **Section 4.3**

Profiles

Java-based Configuration
Integrating Configuration Types
Bean Scope and Lifecycle
Externalizing Properties
Profiles

Profile Overview

- Profiles support environments with different configurations
  - e.g., We need to test a new Catalog service
    - And don't want to corrupt production data
  - So we configure a development DB and datasource
    - With a URL for **test data**
  - When ready, we change to production DB
    - With a different URL for **live data**
  - Giving two desired configurations, as shown below

- **Development** Configuration:
  - An in-memory test database

- **Production** Configuration:
  - The live [1] production database

The First Configuration

- `SpringProdRepositoryConfig` is our production config
  - Uses `@Configuration` style and defines our production repository (1)
  - Later, we'll autowire this into another bean by type

```
@Configuration
@Profile("prod")
public class SpringProdRepositoryConfig {

   @Autowired
   private Integer maxSearchResults;

   @Bean // for this profile, we will use the production repository
   ItemRepository itemRepository() {
      ProductionItemRepository repository =
         new ProductionItemRepository();
      repository.setMaxSearchResults(maxSearchResults);
      return repository;
   }
}
```

The Second Configuration

- `SpringDevRepositoryConfig` is our development config
  - Similar to `SpringProdRepositoryConfig`
  - Just configures a different (in-memory) repository

```
@Configuration
@Profile("dev")
public class SpringDevRepositoryConfig {

   @Autowired
   private Integer maxSearchResults;

   @Bean // for this profile, we will use the in-memory repository
   ItemRepository itemRepository() {
      InMemoryItemRepository repository =
         new InMemoryItemRepository();
      repository.setMaxSearchResults(maxSearchResults);
      return repository;
   }
}
```

No Change in the Wiring

- `CatalogImpl` is a dependent class
  - Needs an `ItemRepository` - injected into `SpringServiceConfig`
  - Will be either `InMemoryItemRepository` or `ProductionItemRepository` (1)

```
@Configuration
public class SpringServicesConfig {

   @Autowired  // here the configured repository is injected (1)
   private ItemRepository repository;

   @Bean
   Catalog catalog() {
      CatalogImpl cat = new CatalogImpl();
      cat.setItemRepository(repository);
      return cat;
   }
}
```

Specifying Supported Profiles - @Profile

- `@Profile` specifies supported profile(s) (shown at bottom)
  - A configuration is registered if any one of its profiles is active
- Specify **multiple profiles** like this:

  `@Profile({"dev", "other"}) // Use {} for array` (1)
- `@Profile("default")` specifies the default profile
  - Active if no other profile is active
- You can also apply `@Profile` to methods (2)

```
@Configuration
@Profile("dev") // Supports the dev profile only
public class SpringDevRepositoryConfig { /* … */ }
```

```
@Configuration
@Profile("prod", "default") // Supports the prod and default profiles
public class SpringConfig { /* … */ }
```

Enabling Profiles

- Easy programmatically - as shown at bottom
  - **`dev`** profile enabled - **`SpringDevRepositoryConfig`** is used
- You can also set the **`spring.profiles.active`** property
  - Through system environment variable, JVM system property, etc.
  - e.g. to enable the **`dev`** profile(s) for a JVM property (1)

    `-Dspring.profiles.active=dev`
- Multiple profiles may be enabled, e.g.

```
ctx.getEnvironment().setActiveProfiles("dev", "other");
```

```
AnnotationConfigApplicationContext ctx =
    new AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("dev");
ctx.register(SpringConfig.class);
ctx.refresh();
```

configuration class must be loaded/registered(2)

Profiles - XML Configuration

- XML configuration is relatively straightforward
- Can do it in separate config files, as shown below
- Can do it in one file with nested <beans> as shown at bottom

```
<beans profile="prod" … >
   <!-- Production bean definitions - details omitted … -->
</beans>
```

```
<beans profile="dev" … >
   <!-- Dev bean definitions - details omitted … -->
</beans>
```

```
<beans  … >

  <beans profile="prod"> <!-- Detail omitted … --> </beans>

  <beans profile="dev"> <!-- Detail omitted … --> </beans>

</beans>
```

Profiles in JUnit tests

- Can specify configuration classes and profiles at **test class level**
  – Applies to all test methods in the test class

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfig.class)
@ActiveProfiles("dev")
public class CatalogTest {
  @Autowired
  private ApplicationContext ctx;
  // remaining detail omitted
}
```

- Can specify config classes and profiles at **test method level**

```
   @Test
   public void testCatalogPositive() {
      AnnotationConfigApplicationContext ctx =
             new AnnotationConfigApplicationContext();
      ctx.getEnvironment().setActiveProfiles("dev");
      ctx.register(SpringConfig.class);
      ctx.refresh();
   }
```

LEARNINGPATTERNS

Lab 2.3: Profiles

In this lab, we will use profiles to customize our definitions for different environments

Review Questions

- How does Java-based configuration differ from XML-based configuration?
- How do singleton and prototype Spring beans differ?
- What types of bean lifecycle events can be associated with callback methods?
- How does Spring Expression Language (SpEL) simplify application development?
- What are profiles, and how can a Spring application be configured with different profiles?

Lesson Summary

- **Java-based configuration** uses **`@Configuration`** classes
  - Uses **`@Bean`** on methods that are bean definitions
  - Lets you include related Java code in config class
  - Same capabilities as XML, but safer and more tooling available
- Can **`@Import`** one `@Configuration` class from another
  - Or **`@ImportResource`** an XML configuration
- In XML, just declare `@Configuration` class as a bean
- Spring easily defines both **singleton** and **prototype** scope beans
  - **Singleton bean**: Single bean instance for a given bean definition
  - **Prototype bean**: New instance for each request of a bean
- Spring beans have a **sophisticated lifecycle**
  - Can interact with creation/destruction fairly easily
  - Many lifecycle steps with fine-grained access (rarely needed)

Lesson Summary

- Spring's **property sources** and **property placeholders** support externalizing values, and injecting them via **`${propName}`** values
  - **SpEL** is an expression language for bean access and manipulation (and much more)

- **Profiles** support environments with different bean configurations
  - To easily vary configuration for different environments
    - e.g. development or production
  - A configuration is only registered if its **profile is active**
  - Profiles are useable in both `@Configuration` and XML config

Session 3: Introduction to Spring Boot

maven Overview

Spring Boot Overview

Spring Boot Hello World

Lesson Objectives

- Become familiar with maven
- Understand the capabilities of Spring Boot
- Be familiar with the Spring Boot Structure
- Create and run a simple application using Spring Boot

maven Overview

maven Overview

Spring Boot Overview

Spring Boot Hello World

About Maven

- **Maven**: Tool for building and managing Java projects
  - **POM** used for configuration (Project Object Model - an XML file)
  - Maven automates the build process (and much more)

- Maven can **automatically download dependencies**
  - As configured in the POM, and downloaded as part of build
  - Default: Download via http from the **maven central repository**

    http://repo.maven.apache.org/maven2/

- **Gradle**, another build tool, provides similar support
  - Uses Groovy for configuration - details beyond scope of course

- Maven or Gradle are **de-facto standards for Spring** builds
  - Well supported by Spring
  - Easily manage Spring dependencies (see notes)

How We'll Work With Maven

- **Goal**: Understand how maven supplies Spring dependencies
  - Core to Spring Boot's operation
  - Will NOT give an in-depth coverage of maven
  - Will not cover Gradle at all
- We'll mainly use maven for **dependency management**
  - From within your IDE, which we also use for building and running
  - Labs include all details
  - We don't cover maven's other (extensive) capabilities
- maven will be configured in a **special way** in labs
  - To find dependencies locally
  - Reduces the chance of firewall or security settings problems

Core Maven Concepts

- We'll cover many of these in more detail
- **Project**: Central organizing structure
  - Everything you build is in a project
- **POM** (Project Object Model): The configuration of the project
  - The metadata (generally XML)
- **Artifact**: Something produced or used by a project
- **Dependency**: An external artifact needed by a project
- **Plug-In**: Used by maven to implement functionality
  - Many plugins are part of maven, and many third-party plugins
- **Repository**: Stores artifacts

The POM (Project Object Model)

- **XML file** with details about a maven project
  - Typically in root of project, and named ***pom.xml***
- Below, we illustrate POM structure
  - Includes some **required POM elements**
  - Illustrates a **dependency** on a Spring module
  - We'll cover the details next

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javatunes.spring</groupId>
  <artifactId>Lab03.5</artifactId>
  <version>1.0</version>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.0.4.RELEASE</version>
    </dependency>
  </dependencies>
</project>
```

POM - Required Elements

- The following POM elements are required
  - **project**: The root element
    - It should include namespaces (omitted in the example)
  - **modelVersion**: The **maven** model version
    - Should be 4.0.0 for current versions of maven
  - **groupId**: The id of your project's group
  - **artifactId**: The id of the project
  - **version**: The version of the project
  - **groupId:artifactId:version** uniquely identifies the project
    - This is called the fully qualified artifact name

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javatunes.spring</groupId>
  <artifactId>Lab01.1</artifactId>
  <version>1.0</version>
</project>
```

POM - External Dependencies

- This POM declares one external dependency
  - On **spring-context** (A core Spring artifact)
    - Referenced via its fully qualified artifact name
    - Note the property for the Spring version (to change it easily)
  - With this POM, maven **automatically includes** the Spring context jar, **and its dependencies**, in your project

```
<project> <!-- Previous detail omitted -->

  <properties>
    <org.springframework.version>5.0.4.RELEASE</org.springframework.version>
  </properties>

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

</project>
```

Common Maven Artifacts for Spring

- Below are some common Spring artifacts
  - They have GroupId `org.springframework`
  - Can be used as dependencies in POMs

- **spring-aop**: Aspect Oriented Programming (AOP) Framework
- **spring-beans**: Bean Factory and JavaBeans utilities
- **spring-context**: Application Context
- **spring-core**: Core utilities used by other modules
- **spring-expression**: Expression Language
- **spring-instrument**: Instrumentation for AOP
- **spring-jdbc**: JDBC Data Access Library (e.g. `JdbcTemplate`)
- **spring-orm**: Object-to-Relation-Mapping (ORM) integration e.g. JPA
- **spring-test**: Support for unit testing
- **spring-tx**: Spring transaction management
- **spring-web**: Web application development utilities
- **spring-webmvc**: Spring MVC for Servlet Environments

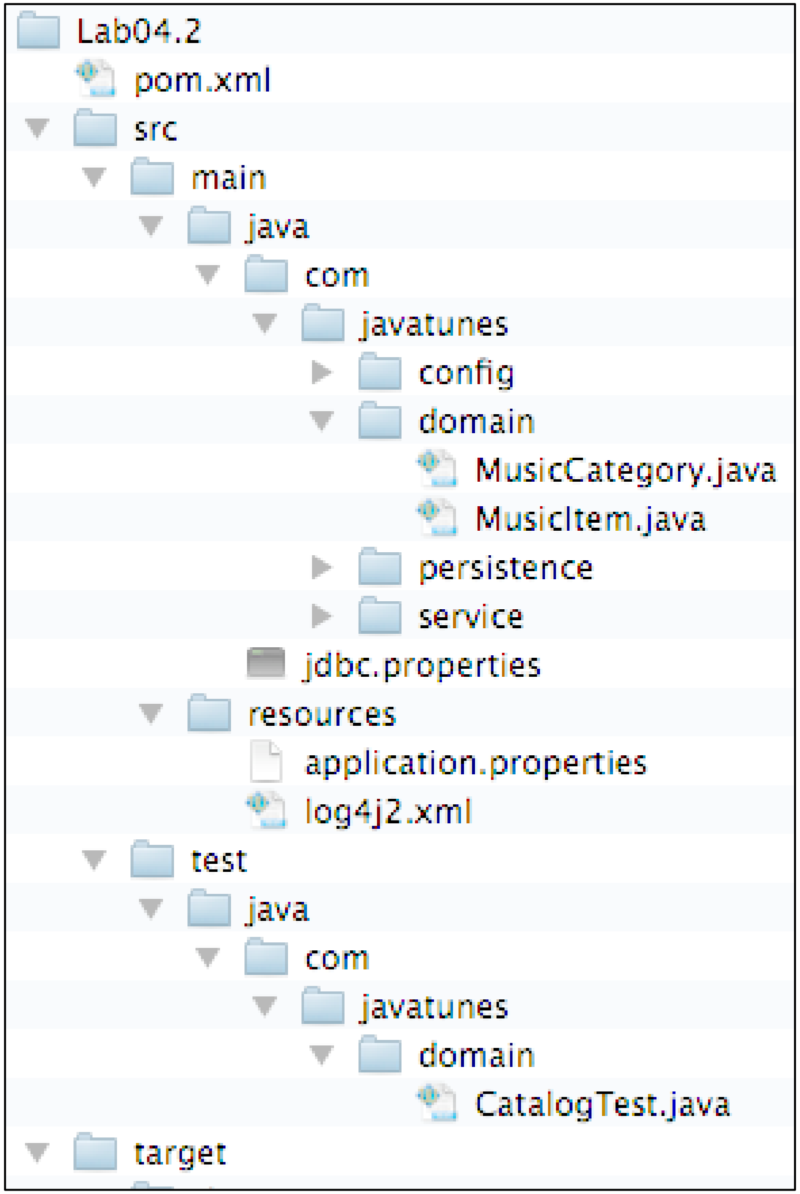
Repositories

- **Repository**: Stores artifacts
  - Maven gets needed artifacts (dependencies) from a repository
- Maven looks first in your **local repository**
  - Default location: ***~/.m2/repository***
- Artifacts not in the local repository are downloaded from a **remote repository**
  - Then **copied to your local repository**
  - Default remote repository, "The Maven Central Repository" at: ***http://repo.maven.apache.org/maven2/***
  - Searchable online at **http://search.maven.org/**
- You can configure this differently
  - Details are beyond the course scope

Maven Java Project Structure

- Standard structure for standalone Java project
  - **Root folder**: *pom.xml*
  - ***src/main/java***: Java source files
  - ***src/main/resources***: Resources on the classpath (e.g. config files like *applicationContext.xml*)
  - ***src/test***: Test source files
  - ***target***: Build output

- Web projects have an additional folder - for JSPs, html, etc
  - ***src/main/webapp***



Executing maven Goals

- At bottom, we execute the **`compile`** goal (leaving out downloading)
  - Maven program is called **`mvn`**
    - Assume `mvn` is on your classpath
    - Assume *pom.xml* is in the current directory
  - When this command is run, you may see lots of output
  - Telling you what `mvn` is doing - e.g. downloading dependencies

```
> mvn compile
[INFO] Scanning for projects...
Downloading:
https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-
starter-parent/2.0.0.RELEASE/spring-boot-starter-parent-2.0.0.RELEASE.pom
… much mvn logging omitted
[INFO] ------------------------------------------------------------------
[INFO]  Building Simple Spring Hello World 1.0
[INFO] ------------------------------------------------------------------
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) …
[INFO] Compiling 3 source files to …
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------------------------------
```

Spring Boot Overview

maven Overview

Spring Boot Overview

Spring Boot Hello World

Motivation for Spring Boot

- Spring: A "lightweight solution" for Java enterprise applications
  – But lightweight is relative [1]

- Spring systems may include Web (MVC/REST), persistence/ORM, AOP, transactions, etc.
  – Even "lightweight" solutions **can take significant effort**

- Spring Boot **reduces the programming and configuration** required to use Spring
  – Sometimes abbreviated **SB** in the course

Spring Boot Project

- **http://projects.spring.io/spring-boot/**
- Framework easing creation of production-grade Spring apps
  - Takes an **opinionated view** of Spring and third-party libraries
- **Opinionated**: What's that?
  - Reduces configuration by making choices for you (the opinions)
  - i.e. favors **convention over configuration** (COC)
- Some "opinions" it has:
  - **Spring modules** to include: e.g. always use Spring core
    - Or "always use Spring JDBC/JPA and a connection pool when a DB is detected"
  - **What Versions**: (e.g. Spring 5.0.4 or Tomcat 8.5.24)
  - **Automatic configurations**: e.g. auto-configure the H2 DB if an H2 build dependency is detected

Goals for Spring Boot Project

- Provide a **radically faster and widely accessible** getting starting experience for Spring development [(1)]
  - Easily create production-ready Spring apps
- **Be opinionated**, but **get out of the way quickly** as requirements differ from defaults
  - Convention over configuration
  - Easy to change: e.g. specify Spring 5.0.3 instead of 5.0.4
- Provide common **non-functional** features
  - Embedded servers, security, metrics, health checks, externalized configuration, etc.
- No code generation nor requirement for XML configuration

How is Spring Boot Structured?

- It's a set of **jar files** and **dependency declarations**
  - Dependencies use maven POM format (or gradle)

- **Starter POMs** provide functionality in different areas
  - Conveniently packaged for ease of use
  - e.g., the JPA starter adds all you need to use JPA
    - All dependencies and a bunch of auto-configuration
    - Provides sensible defaults that often work

- Spring Boot works with a build management system
  - maven or gradle
  - We'll use **maven** in the course
  - There is also a Command Line Interface (CLI) for interactively developing with Spring (Groovy-based)

Dependency Management with Spring Boot

- Projects declare dependencies on **Spring Boot artifacts**
  - **Not** directly on Spring artifacts
- Below, is a maven POM using Spring Boot (1)
  - Declares its **parent POM** as a special **Spring Boot starter parent**
  - Specifies a dependency on a starter: **`spring-boot-starter`**
  - Much more detail on all of this soon

```
<project … >  <!-- Namespaces/other detail omitted -->
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version> <!-- See note (2) on version -->
  </parent>
  <artifactId>spring-boot-helloWorld</artifactId>

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

Equivalent POM without Spring Boot

- We show the interesting parts (leaving out some basic detail)
  - Full one doesn't even fit in slide - which do you want to write/maintain?

```
<project … >  <!-- Namespaces/other detail omitted -->
  <artifactId>spring-helloWorld</artifactId> <!-- Detail omitted -->

  <properties>
    <spring.version>5.0.4.RELEASE</spring.version>
    <!-- More version properties omitted to save space
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>
    <!-- Dependencies for log4j, javax.inject not shown -->
  </dependencies>
</project>
```

Configuration with Spring Boot

- Projects benefit from Spring Boot **autoconfiguration**
  - Reducing (or eliminating) standard Spring configuration
- Below is a simple app enabling autoconfiguration
  - Via **`@EnableAutoConfiguration`** - a Spring Boot annotation
  - Enables all of Spring Boot's auto-config capabilities
  - Can still use any of Spring's configuration explicitly
    - Overrides any autoconfiguration

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.SpringApplication;

@Configuration
@EnableAutoConfiguration
public class DemoApplication {
  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);  // Covered next
  }
}
```

Programming with Spring Boot

- Spring Boot provides convenient shortcuts - one is shown below
  - You can also program to standard Java/Spring
- **`SpringApplication.run()`** does the following:
  - Creates an `ApplicationContext` configured by the passed in sources
    - In this case, our `DemoApplication` config class
  - Exposes command line arguments as Spring properties
  - Loads all singleton beans into the context
  - Triggers any beans implementing `CommandLineRunner` (covered later)

```
import org.springframework.boot.SpringApplication;  // Other imports omitted

@Configuration
@EnableAutoConfiguration
public class DemoApplication {

    public static void main(String[] args) {
     System.out.println("DemoApplication.main() called");
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Getting Started

- The reference manual and API (javadoc) docs
  - Links at project page: *http://projects.spring.io/spring-boot/*
  - Invaluable, but more is needed to ramp up

- Sample projects
  - On github: ***https://github.com/spring-projects/spring-boot/***
    - Look for the spring-boot-samples under that

- Project generator page
  - ***http://start.spring.io***
  - Generates configurable starter projects
  - Called the **Spring Initializr**

- Next up: A simple Spring Boot-based project

Mini-Lab: Review Boot Reference

Mini-Lab

- We provide the Spring Reference Reference under
  - Under *StudentWork/Spring/**Resources/SpringBootDocs/reference***
  - Spend five minutes looking at it as described below

- In a browser, open *index.html* in the folder above
  - Briefly look at the "**Introducing Spring Boot**" section
    - Currently **Section 8**
  - Briefly look at the "**Creating the POM**" section
    - Currently **Section 11.1**
  - If you have any time left look at any other sections of interest

Mini-Lab: Spring Boot - Starter Projects

- We'll generate some projects using the Initializr

Mini-Lab

- Browse to: ***start.spring.io***
- Click **Generate Project** (take all the defaults)
  - Save the file as *demo-simple.zip* (some place it's easy to get to)(1)
  - Unzip the archive, and review it
- Type **Web** in the Dependencies search text box
  - Add in the full-stack Web development with Tomcat
  - Generate the project again, unzip, and review
- Try any of the other technologies available

Spring Boot Hello World

maven Overview

Spring Boot Overview

Spring Boot Hello World

Our Goals for Hello World

- **Illustrate a complete** Spring Boot app
  - To familiarize ourselves with using Boot
  - The app does very little - so Boot's benefits are small
  - Latter apps will benefit more from Boot

- **Dive into** Spring Boot's magic
  - Peek under the hood to better understand what it does
    - To make better choices on using it
    - To use it correctly

POM for Hello World

- Our previously illustrated POM:
  - Declares its **parent** POM to be **`spring-boot-starter-parent`**
  - So inherits all the parent configuration (1)
  - Specifies a dependency on a Boot starter: `spring-boot-starter`
- Let's review how Spring Boot works with this POM (2)

```
<project … >  <!-- Namespaces/other detail omitted -->
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.0.0.RELEASE</version>
   </parent>
   <artifactId>spring-boot-helloWorld</artifactId>

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

spring-boot-starter-parent

- Special starter with sensible/useful defaults including:
  - Java 8 (1.8) default compiler level (Required by Spring 5)
  - UTF-8 source encoding
  - Dependency Management section declaring **default versions**
    - You can omit `<version>` for many dependencies - like Spring
    - Version numbers are declared in `spring-boot-dependencies` - the parent of `spring-boot-starter-parent`
  - Sensible resource filtering and plugin configuration (1)

- This starter is just a single *pom.xml* file
  - Excerpts illustrated on next slide
  - No jar files
  - Consider this the "**`parent`**" starter project (2)

spring-boot-starter-parent POM Excerpt

- Brings in **spring-boot-dependencies** (its parent)
  - Which mainly declares default versions
- Declares several properties
  - e.g. Java compiler version (using a Spring Boot property)
- We'll look at the actual POM file shortly

```
<parent>  <!-- Bring in spring-boot-dependencies configuration -->
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>${revision}</version>   <!-- Defined in root project -->
</parent>                          <!-- as 2.0.0.RELEASE -->

<properties>  <!-- Samples - other properties omitted -->
  <java.version>1.8</java.version>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

spring-boot-dependencies POM Excerpt

- Below, we show some property definitions to see how it works
  - As well as some dependencies in <dependencyManagement> (1)
  - You generally never need to open/work with this file

```
<properties>  <!-- Samples - Most properties omitted -->
  <derby.version>10.14.1.0</derby.version>
  <tomcat.version>8.5.28</tomcat.version>
  <spring.version>5.0.4.RELEASE</spring.version>
</properties>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot</artifactId>
      <version>${revision}</version> <!-- 2.0.0.RELEASE -->
    </dependency>

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

spring-boot-starter

- The core Spring Boot starter
  - Consists of just a *pom.xml*
  - Brings in core capabilities, auto-configuration, logging , YAML
    - Also brings in **`spring-core`**
  - Many SB starters depend on `spring-boot-starter`
  - Recall that Hello World declared a dependency on this
- Dependency on a starter is generally required in your POM
  - `spring-boot-starter-parent` as a parent isn't enough
    - It just sets some defaults (e.g. software versions)
  - `spring-boot-starter` actually brings in dependencies
- We'll briefly review some POM excerpts

spring-boot-starter POM Excerpt

- Below we illustrate some dependencies it declares
  - Spring Boot and Spring Core
  - It's a straightforward maven POM

```
<dependency>  <!-- Version elements omitted below -->
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-autoconfigure</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-logging</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-core</artifactId>
   <exclusions> <!-- Detail omitted -->  </exclusions>
</dependency> <!-- Remaining detail omitted -->
```

Summary

- Spring Boot is structured as a collection of maven projects
  - We've taken a look at some of the core projects/POMs and their dependencies
- **`spring-boot-starter-parent`**
  - Special parent POM providing useful defaults
- **`spring-boot-dependencies`**
  - Declares default versions
  - Brought in by `spring-boot-starter-parent`
- **`spring-boot-starter`**
  - Core Spring Boot starter
  - Brings in the core Spring Boot projects, as well as Spring core

Mini-Lab: Spring Boot - Brief Review of POMs

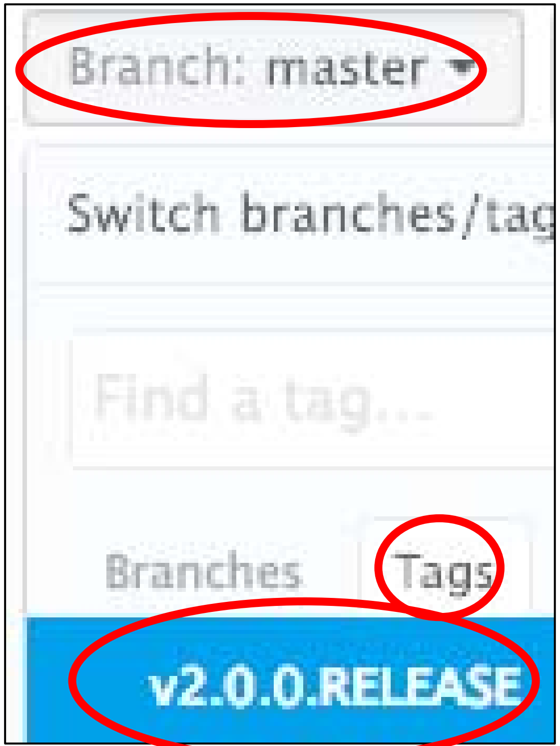
- We'll examine some of the POMs we've seen excerpts of

Mini-Lab

- Browse to:

https://github.com/spring-projects/spring-boot

  - Go to Branch dropdown, select **tag** v2.0.0.RELEASE
    - Go to ***spring-boot-project*** - Starter page for the below
- Browse to *spring-boot-dependencies/pom.xml*
  - It mainly defines default versions for the pieces needed for Spring Boot
- Browse to *spring-boot-starters/spring-boot-starter-parent/pom.xml*
  - It brings in `spring-boot-dependencies` as its parent and defines some properties
- Browse to *spring-boot-starters/spring-boot-starter/pom.xml* (1) (brings in a few base dependencies)



Hello World Overview

- We'll show a simple Spring Boot-based app
  - That uses a `main()`
- Uses some Boot glue code to boot the app
- Uses some Boot API to code its functionality
  - For convenience
- These capabilities are part of **`spring-boot`**
  - Brought in by `spring-boot-starter`

Hello World Code

- **@SpringBootApplication** declares an `@Configuration` class appropriate for Spring Boot (1)
- **SpringApplication.run()** boots our app using this config class
  - run() is called auto-magically - more details later

```
// Some imports/detail omitted
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class HelloBootWorld implements CommandLineRunner {

  public static void main(String[] args) {
    System.out.println("HelloBootWorld.main() called");
    SpringApplication.run(HelloBootWorld.class, args);
  }

  @Override
  public void run(String... arg0) throws Exception {
     System.out.println("HelloBootWorld.run");
  }
}
```

Output of HelloBootWorld

- Below, we see the output
  - It includes output from `main()`, a standard Spring Boot banner, logging messages, and output from `run()`
  - We omit most of the logging

```
HellowBootWorld.main() called

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::            (v2.0.0.RELEASE)

2018-02-25 10:15:44.717  INFO 21619 --- [           main]
com.example.HellowBootWorld    : Starting HellowBootWorld …
// logging omitted.
HellowBootWorld.run
// logging omitted
```

How it Works

- This small app uses many Spring Boot capabilities
  - **Dependencies** are pulled in via Boot POMs
  - **Auto-configuration** is done based on our system
    - Very little in this app - but enough to demo it
  - **App-specific configuration** easily added to `HelloBootWorld`
    - Which is our root configuration class
    - It can pull in other configuration
  - We **kick off app code** with the convenient `CommandLineRunner`

- We'll use this app in the lab to get a feel of Spring Boot
  - We'll see and use more capabilities soon

Easy Configuration with Properties

- Boot supports a lot of configuration via properties files
  - Default - *application.properties* file on the classpath
  - Below, we show logging configuration in that file
  - Configures core spring to level debug and spring boot to warn

- There are many properties
  - We'll mention useful ones in the course, as we encounter the relevant areas

```
logging.level.org.springframework=debug
logging.level.org.springframework.boot=warn
```

Executable Jar Packaging

- Spring Boot can also package applications in a jar
  - For maven, activated by adding **spring-boot maven** in POM
  - Maven Usage illustrated at bottom

- The plugin automatically creates an executable jar
  - When running the **package** goal (a standard maven goal)
    - Modifies the standard goal to create the executable jar
    - Packages up dependencies, and can locate a main class (1)

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

LEARNINGPATTERNS

Lab 3.1: Hello Boot World

In this lab, you will configure and run a simple Spring Boot job

Recap of what we've seen

- We've provided an overview of Spring Boot
  - The core concepts: **POMs**, **Starters, Dependency Management, Auto-Configuration**
  - We looked at a simple **Spring Boot application**

- Some other things Spring Boot can do
  - Auto configuration (web apps, datasources, etc.)
  - Monitoring/metrics of applications

- **Resources**
  - **http://projects.spring.io/spring-boot/**
    - The home of the Spring Boot Framework
  - The **Spring Boot documentation**
    - The reference manual, API docs, and samples
    - Available on the project home

Session 4: Spring Testing

Testing and JUnit Overview

Spring TestContext Framework

Lesson Objectives

- Review testing and JUnit Basics
- Learn about and use Spring Testing's integration between JUnit and Spring
- NOTE: We use **JUnit 4** in the course
  - Not JUnit 5, which is the latest release
  - JUnit 4 is still much more widely adopted than JUnit 5

Testing and JUnit Overview

Testing and JUnit Overview

Spring TestContext Framework

May skip this small section if comfortable with JUnit,
and go on to Spring TestContext Framework

Testing Overview

- Testing: Critical part of software development
  - Assess software **behavior** with respect to **expectations**

- **Unit testing** focuses on "individual units" of a larger system
  - Typically a **method** of a class
  - ***Test cases***: Test methods to perform unit tests
    - Shows that code does what we (developers) expect
  - Verifies **correctness** of individual parts of a system
    - And that they **remain correct** when **changes** are made (**this is key**)

- **Integration testing**: Test software modules as a group
  - More applicable when talking about Spring
  - Spring is responsible for creating/wiring objects
  - Makes sense to do integration testing with Spring doing its work

JUnit Overview

- Open-source Java testing framework
  - Most popular in the industry
  - Universally supported in IDEs and build tools
- **Automates** testing of Java code
- Provides standard way to write and organize tests
- Consists of classes and annotations to write and run tests
  - **Annotations** for declaring test methods (`@Test`)
  - **Assertions** to test actual results against expected results
  - **Test fixtures** to set up each test's environment
  - **Test runners**
    - Several provided, plus 3rd party runners

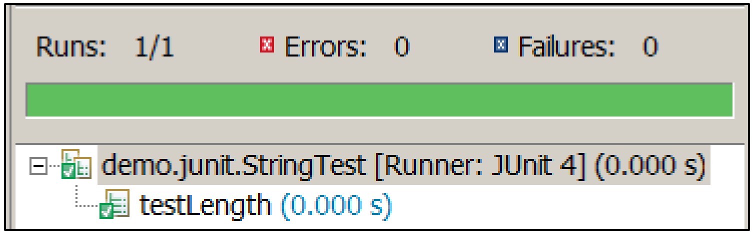
Writing a Test – First Example

- Create test class
  - Write test methods (annotated with **`@Test`**)
  - Set up business object, invoke business method on it
  - Make assertions about results via **static** methods in **`Assert`**
  - JUnit base package is **`org.junit`**

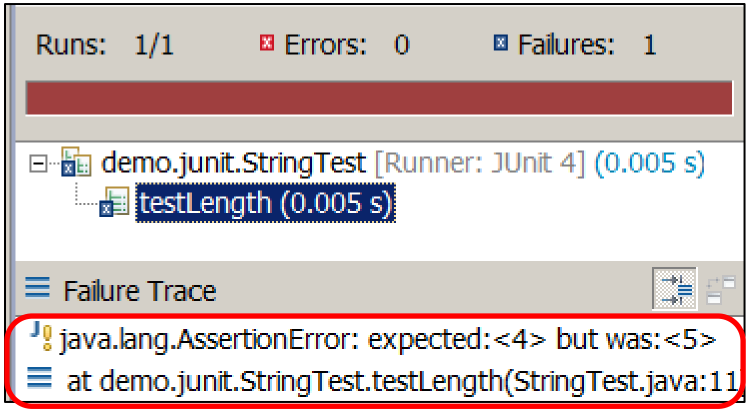
```
package demo.junit;

import static org.junit.Assert.*;   // static import typically used
import org.junit.Test;

public class StringTest {

  @Test
  public void testLength() {
    String msg = "hello";             // object under test
    assertEquals(5, msg.length());   // expected, actual
    assertTrue(5 == msg.length());   // alternative assertion
  }
}
```

Running Tests in the IDE

- All modern IDEs provide a graphical test runner
  – Usually provide a progress bar: green = pass, red = fail
  – ***"If the bar is green, the code is clean"***



- Failed tests offer "advice" as to why it failed, and where



Running Tests in Other Environments

- Ant **`<junit>`** task in *build.xml*

  `> ant run-tests`

- Maven test phase

  `> mvn test`

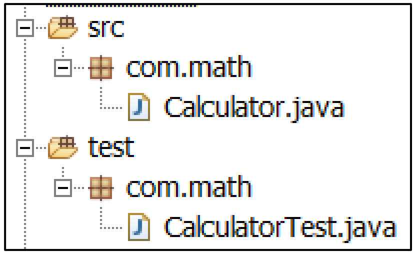
- Standalone (command line)
  - Use `org.junit.runners.`**`JUnitCore`** (has a `main` method)

- Programmatically (by invoking `JUnitCore`)
  - See notes for details

- 3rd party runners, e.g., `SpringJUnit4ClassRunner`
  - Sets up test client using Spring-injected objects

Naming Conventions and Organizing Tests

- `MyClass` is tested by `MyClassTest`
- `MyClassTest` is in **same package** as `MyClass`
- Parallel ***src*** and ***test*** directories
  - Both branches are checked into source control
  - For packaging / deployment, only the *src* classes



- For test methods:
  - `myMethod()` is tested by `testMyMethod()`
  - **Must** be annotated by `@Test` to be run as a test case
    - Method name technically doesn't matter, but this historical naming convention still widely used (see notes)

Positive and Negative Tests

- **Positive** tests test for behavior that **works as expected**
  - e.g. that with **valid input** you get **valid results**
  - They can be named with the suffix **`Positive`**
  - e.g. **`testMyMethodPositive()`**

- **Negative** tests test for behavior that **doesn't work as expected**
  - e.g. Invalid input gives expected results (e.g. an exception)
    - Which can be indicated with the **`expected`** element to `@Test`

    `@Test(expected = NullPointerException.class)`
  - They can be named with the suffix **`Negative`**
  - e.g. **`testMyMethodNegative()`**

- We'll see an example of this in the lab

Writing Test Methods

- Write a method in the test class that is:
  - Annotated with **`@Test`** (required)
  - **`public`**
  - **`void`** return type
  - Convention: method name starts with **"test"** (not required)
    - Be descriptive! Long method names are okay!
- In the test method body:
  - Set up the target object and invoke the target method
    - Using varying arguments, verify return values and/or side effects
- Order of test execution **cannot be predicted or relied upon**
  - Don't write test methods depending on other test methods running first (an **anti-pattern**)

Assertions

- State your expectations – as static method calls on `Assert`
  - All overloaded with a variant that takes a string message

- `assertEquals(expected, actual)`

- `assertTrue(should-be-true-condition)`
  `assertFalse(should-be-false-condition)`

- `assertNull(object-that-should-be-null)`
  `assertNotNull(object-that-should-not-be-null)`

- `assertSame(object1, object2)` ==
  `assertNotSame(object1, object2)` !=

- `assertArrayEquals(expecteds[], actuals[])`

Test Fixtures – @Before and @After

- Often, several tests operate on the same set of target objects
  - You CAN set these objects up in each test method
  - BUT this setup code is usually redundant (bad), and not related to **testing** those objects (which should be the test's focus)

- ***Test fixtures*** solve this problem
  - Add private fields for each target object you need
  - Write a "setup" method annotated with **`@Before`** to initialize them
  - If needed, write a "cleanup" method annotated with **`@After`**

- It's all about **eliminating redundancy**
  - Put **common** setup / cleanup code in the fixture methods
  - Put **test-specific** setup code in the test method itself

Test Fixtures – Example

- The **msg** variable is initialized before **each** test method

```
public class StringTest {        // package and imports not shown
  private String msg;

  @Before
  public void init() {           // see notes for naming conventions
    msg = "hello";
  }

  @Test
  public void testLengthPositive() {
    assertEquals(5, msg.length());
  }

  @Test
  public void testSubstringPositive() {
    assertEquals("he", msg.substring(0, 2));
  }
}
```

Test Fixtures – `@BeforeClass` and `@AfterClass`

- Use **`@BeforeClass`** and **`@AfterClass`** for one-time or "expensive" setup / cleanup
  - **NOTE**: these methods must be declared **`static`**

- `@BeforeClass` method runs **once**, before any tests
- `@AfterClass` method runs **once**, after all the tests

- Objects created in `@BeforeClass` are **not cleaned up** in any way after each test is run
  - Only when all tests are complete – in the `@AfterClass` method
  - **Potentially compromises the independence of tests**
    - Should be used judiciously, if at all
    - Consider using mock or fake objects instead, initialized in `@Before`

Order of Execution

- `@Before` method runs before **each** test method
- `@After` method runs after **each** test method
- Each test method is "bracketed" with these methods

- **Entire** test run lifecycle is bracketed by `@BeforeClass` and `@AfterClass` methods

```
 1              2                    3
@BeforeClass  @Before    @AfterClass
    @Test  testOne()
    @After
    @Before
    @Test  testTwo()
    @After
```

[Optional] Lab 4.1: Using JUnit

In this lab you will set up and run JUnit tests

You can skip this lab if comfortable with JUnit

LEARNINGPATTERNS

Spring TestContext Framework

Testing and JUnit Overview

Spring TestContext Framework

Spring TestContext Overview

- Comprehensive support for integration testing, including:
  - **Loads the Spring configuration**, and initializes container
    - Does normal Spring initialization based on config
    - Can use same config as your system uses
  - Supports **dependency injection** into test classes
  - Additional support, including:
    - DB interactions and transactions
    - Out-of-container Web app testing

- Maven dependencies (as `groupId:artifactId`)
  - Standalone: **`org.springframework:spring-test`**
  - Spring Boot:
    `org.springframework.boot:spring-boot-starter-test`

Common TestContext Types

- Class `SpringJUnit4ClassRunner`/`SpringRunner`
  - Custom JUnit 4 runner that enables the Spring TestContext
  - package: `org.springframework.test.context.junit4`
  - `SpringRunner` is an alias class you can use

- `@ContextConfiguration`
  - For Spring metadata loading (XML or `@Configuration`)
    - Can only specify one type of metadata (1)
  - Package: `org.springframework.test.context`

- `@ActiveProfiles`: Set the profiles to be used
  - Package: `org.springframework.test.context`

Example: Using Spring's TestContext

- Below, is a JUnit test using the TestContext support
  - Note how we boot the container, and inject the Catalog
  - At bottom, we do this without the TextContext support

```
// imports, other detail omitted
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={SpringConfig.class})
public class CatalogTests {

  @Autowired Catalog cat;

  @Test testCatalogFindById() { /* Use cat as needed */ }
}
```

```
public class CatalogTests {  // imports, other detail omitted
  @Test testCatalogFindById() {
    AnnotationConfigApplicationContext ctx =
      new AnnotationConfigApplicationContext(SpringConfig.class);
    Catalog cat = ctx.getBean(Catalog.class);
    /* Use cat as needed … */
  }
}
```

Context Management / Caching

- TestContext **caches** the Spring context once loaded
  - The context **is reused** for each test suite
    - Test suite: All tests that run in the same Java JVM
  - Reduces testing startup time in large systems

- Contexts cached based on config specification
  - e.g. key generated from `SpringConfig.class` in our example
  - With multiple config classes, all are used to generate the key

- Cached context is used across test classes as long as in same VM
  - e.g. if run using maven without forking a new process

- Disable selectively via **`@DirtiesContext`** on test class or method
  - Removes associated context after test execution

Using Spring Boot Test

- Spring Boot Test is slightly different in usage
  - Use **`spring-boot-starter-test`** in maven POM
  - Use **`@SprinBootTest`** on test class, not `@ContextConfiguration`
- **`@SpringBootTest`** ties Spring Boot into TestContext framework
  - Loads config classes (via **`classes`** element)
  - Enables Spring Boot's capabilities
  - Supports custom environment properties (via **`properties`** element)
  - Registers REST/Web client beans for web tests going to external server

```
// imports, other detail omitted
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes={SpringConfig.class})
public class CatalogTests {

  @Autowired Catalog cat;

  @Test testCatalogFindById() { /* Use cat as needed */ }
}
```

TestContext: Under the Hood

- Done via types in `org.springframework`**`.test.context`**
  - Useful to know how these work, but not required
  - Needed to customize behavior

- Interface **`TestContext`**: Encapsulates the test context
- Class **`TestContextManager`**: Manages a `TestContext`
  - And fires events to `TestExecutionListeners`
- **`TestExecutionListener`**: Listener API for test events
  - Implementations include **`DirtiesContext`**, **`ServletTest`**, **`DependencyInjection`**, **`TransactionalTestExecutionListener`**
  - The above are registered by default, there are others
  - Listeners run before the tests, and provide DI, TX, etc.
- **`@TestExecutionListeners`**: Supports specifying the listeners for a manager
  - Can omit a default, or add others (e.g. for DBUnit providing DB support)

Lab 4.2: Using Spring Testing

In this lab, you will use some of the Spring Testing Capabilities

Session 5: Database Access With Spring

Overview

Using Spring with Hibernate

Using Spring with JPA

Spring Data Overview

Using Spring Data

Lesson Objectives

- Understand Repositories/DAOs, and how Spring supports them
- Be familiar with Spring's datasource support
- Be familiar with the Spring Support for Hibernate and JPA
  - Note: **The Hibernate and JPA sections are independent**, so you can skip whichever you're not using
    - By using Hibernate we mean using the Hibernate XML mapping, and Session / SessionFactory
    - If you are using the JPA API, but with an underlying Hibernate implementation, you can skip the Hibernate section
  - They assume some familiarity with Hibernate or JPA, but briefly review the core concepts for those who are not familiar with it

Overview

Overview

Using Spring with Hibernate

Using Spring with JPA

Spring Data Overview

Using Spring Data

Data Access Support

- **Repositories** encapsulate DB code
  - Also called a **DAO** (**Data Access Object**) (1)
  - Puts DB functionality **in one place**, making it easier to manage
    - **Encapsulates** DB access - internals not accessible to clients
    - **Insulates other code** from DB details - clients are simpler

- Spring has extensive repository support
  - We focus on Hibernate and JPA-based repositories
  - Not JDBC-based ones - not as widely used any more

- We'll look at Spring's **datasource** support
  - In **`spring-jdbc`**
  - **`spring-boot-starter-jdbc`** pulls in everything needed
  - Including TX support in **`spring-tx`** module

Datasources

- **`javax.sql.DataSource`**: Resource class for DB connections
  - Connection factory for underlying physical DB
  - Provides connections via `getConnection()`
  - Configured with DB connection information

- Spring's `DataSource` related classes include:
  - **`DriverManagerDataSource`**: Simple, non-connection pooled `DataSource` implementation
    - For testing/development
    - Configuration properties include `driverClassName`, `url`, `username`, `password`
  - **`JndiObjectFactoryBean`**: General purpose JNDI look up
    - Can look up an existing `DataSource` in JNDI
    - For example in a JEE environment

Example: Configuring a DataSource

- Below, we configure a `DataSource` (via XML)
  - A Spring **`DriverManagerDataSource`** [1]
  - **`personnelDataSource`** contains DB connection info
- At bottom, we show the Java-based equivalent

```
<bean id="personnelDataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver"/>
  <property name="url"
            value="jdbc:derby://localhost:1527/PERSONNEL"/>
  <property name="username" value="guest"/>
  <property name="password" value="password"/>
</bean>
```

```
@Bean DataSource personnelDataSource() {  //
  DriverManagerDataSource ds = new DriverManagerDataSource();
  ds.setDriverClassName("org.apache.derby.jdbc.ClientDriver");
  // See notes for detail on remaining properties …
  return ds;
}
```

Example: JNDI Lookup of a DataSource

- Especially useful in JEE environments
  - Very simple via the **jee** namespace (below)
  - Earlier versions used `JndiObjectFactoryBean` (1)
- In Java-based config just use JNDI directly (bottom)

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

   <jee:jndi-lookup id="personnelDataSource"
                    jndi-name="java:comp/env/jdbc/personnelDS">

</beans>
```

```
  Context ctx = new InitialContext();
  DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/personnelDS");
```

Properties Files

- Properties files are a good choice for DB connection info
  – Easy to change (for all configuration types)
  – Profiles can also be useful
- Below, are simple property file examples [1]

```
<bean id="personnelDataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <!-- Other properties configured similarly -->
</bean>
<context:property-placeholder location="jdbc.properties"/>
```

```
@Configuration @PropertySource("classpath:jdbc.properties")
public class MyConfig {
  @Autowired Environment env;

  @Bean DataSource javatunesDataSource() {  //
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName(env.getProperty("jdbc.driverClassName"));
    // Other properties configured similarly
    return ds;
}
```

XML vs Java Config vs Properties Files

- **Properties files**: Good choice for DB connection info
  - Especially since you likely use them for other things

- **XML** is a good choice - especially with a single datasource
  - DB info is in one place and easy to change
  - With multiple datasources, properties files may be better

- **`@Configuration`** classes are good with DB info in properties files
  - DB info is separate from the Java config
  - **Not good** with embedded connection info
    - Changes require a recompile, and DB info embedded in a Java class

- Labs use `@Configuration` with DB info in a **properties file**
  - For our straightforward use case, it's the simplest

[Optional] Using Spring with Hibernate (Can skip if not using Hibernate)

Overview
Using Spring with Hibernate
Using Spring with JPA
Spring Data Overview
Using Spring Data

Hibernate Overview

- Hibernate: Open Source, **ORM** (**Object-Relational Mapping**) framework for Java
  - Integrates a **Java OO domain model** with relational data
  - Provides metadata-driven approach to mapping
    - Generally using XML mapping files in classic Hibernate

- Two core configuration files:
  - **Hibernate config file** contains global configuration
    - e.g. **`SessionFactory`** configuration (represents a single DB)
    - Typically called ***hibernate.cfg.xml***
  - **Mapping files** containing class mappings - e.g. ***Employee.hbm.xml***
    - Maps `Employee` class to DB tables

Hibernate Configuration File Illustration

- Includes basic (non-Spring) Hibernate config [1]

```
<!-- Other detail omitted -->
<session-factory>
  <!-- Database Connection Settings -->
  <property name="hibernate.connection.username">guest</property>
  <property name="hibernate.connection.password">password</property>
  <property name="hibernate.connection.url">
     jdbc:derby://localhost:1527/PERSONNEL</property>
  <property name="hibernate.onnection.driver_class">
     org.apache.derby.jdbc.ClientDriver</property>

 <!-- Configure the SQL Dialect for Hibernate -->
  <property name="hibernate.dialect">
     org.hibernate.dialect.DerbyTenSevenDialect</property>

  <!-- Specify a mapping resource -->
  <mapping resource="com/javatunes/persist/Employee.hbm.xml"/>

</session-factory>
```

Using Hibernate Directly

- Two core Hibernate interfaces
  - **SessionFactory**: Contains DB connection info, produces sessions
  - **Session**: Interacts with the DB
  - We show a (very simplified) direct Hibernate example below
- Note: We ignore transactions in the Spring examples that follow
  - We'll get to Spring's transactions later

```
// much code omitted ...
// Create the session factory based on configuration shown previously
SessionFactory sf = new Configuration().configure().buildSessionFactory();
Session s = sf.openSession();  // Create a session
s.beginTransaction();
Employee e = (Employee)s.get(Employee.class,new Long(1));
System.out.println("Retreived Employee: " + e.getName());
s.getTransaction().commit();
s.close();
sf.close();
```

Spring Support for Hibernate

- Spring provides classes to simplify Hibernate programming
  - Layered on type of Hibernate types
  - Simplifies the details of working with them (1)
  - Spring 4 supports Hibernate 3, 4, and 5
  - Spring 5 supports Hibernate 5

- Spring Hibernate packages include:
  - **`org.springframework.orm.hibernate5`**: Main package defining Spring/Hibernate integration
  - **`org.springframework.orm.hibernate5.support`**: Support classes for working with Hibernate

LocalSessionFactoryBean

- **`LocalSessionFactoryBean`**: Spring factory to create `SesssionFactory` instances (1)
  - Provides easy configuration of Hibernate `SessionFactory`
  - A singleton that can be shared

- Properties below support Hibernate configuration:
  - **`dataSource`**: DataSource for the `SessionFactory`
  - **`mappingLocations`**: Hibernate mapping resources, such as *com/javatunes/persist/Employee.hbm.xml*
  - **`hibernateProperties`**: Hibernate specific properties, such as `hibernate.dialect`
  - **`configLocation/configLocations`**: Location of Hibernate XML config file(s) for using Hibernate style configuration

Configuring a Hibernate Session Factory

- See notes for `@Configuration` example

```
<!-- Datasource config not shown - similar to previous config -->
<bean id="personnelSessionFactory"
  class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
  <property name="dataSource" ref="personnelDataSource"/>
  <property name="mappingResources">
    <list>
      <value>com/javatunes/persist/Employee.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
            org.hibernate.dialect.DerbyTenSevenDialect</prop>
    </props>
  </property>
</bean>
```

Contextual Sessions

- Hibernate supports contextual sessions
  - `SessionFactory.getCurrentSession()` provides a session associated with the "current" context
  - Works with JTA (Java Transaction API)
    - Standard JEE Transaction API (generally used in app servers)
    - The session is scoped to a JTA transaction
  - Works with Java SE (standalone programs)
    - Scopes the session to a TX using the thread of execution
- Contextual sessions **ease management and propagation of a Hibernate session**
  - Can write a Hibernate Repository class **with no Spring APIs** (1)
  - But still integrate it with Spring

Example: Spring Free Repository Class

- The example below is Spring free
  – Spring injects the session factory as shown at bottom (XML)
- Can use other Spring capabilities [1]
  – e.g., Spring managed transactions (covered later)

```
package com.javatunes.persist;
import org.hibernate.SessionFactory;

// Interface EmployeeRepository not shown (2)
public class HibernateEmployeeRepository implements EmployeeRepository {
  private SessionFactory sessionFactory; // get/set methods not shown

  public Employee searchById(Long id) {
    return (Employee)sessionFactory.getCurrentSession().get(
       Employee.class, id);
  }
}
```

```
<bean id="hibernateEmployeeRepository"
     class="com.javatunes.persistence.HibernateEmployeeRepository"/>
  <property name="sessionFactory" ref="personnelSessionFactory"/>
</bean>
```

Example: Injecting with @Autowired

- Injecting into `@Configuration` class shown below
- Injecting into component directly at bottom
  - Using Spring's `@Repository` annotation (1)

```
@Configuration
public class RepositoryConfig {
  @Autowired
  private SessionFactory sessionFactory;

  @Bean
  EmployeeRepository hibernateEmployeeRepository() {
    EmployeeRepository rep = new HibernateEmployeeRepository();
    rep.setSessionFactory(sessionFactory):
    return rep;
  }
} // Detail omitted
```

```
@Repository
public class HibernateEmployeeRepository implements EmployeeRepository {
  @Autowired
  private SessionFactory sessionFactory;
} // Most detail omitted
```

Using Spring with JPA

Overview

Using Spring with Hibernate

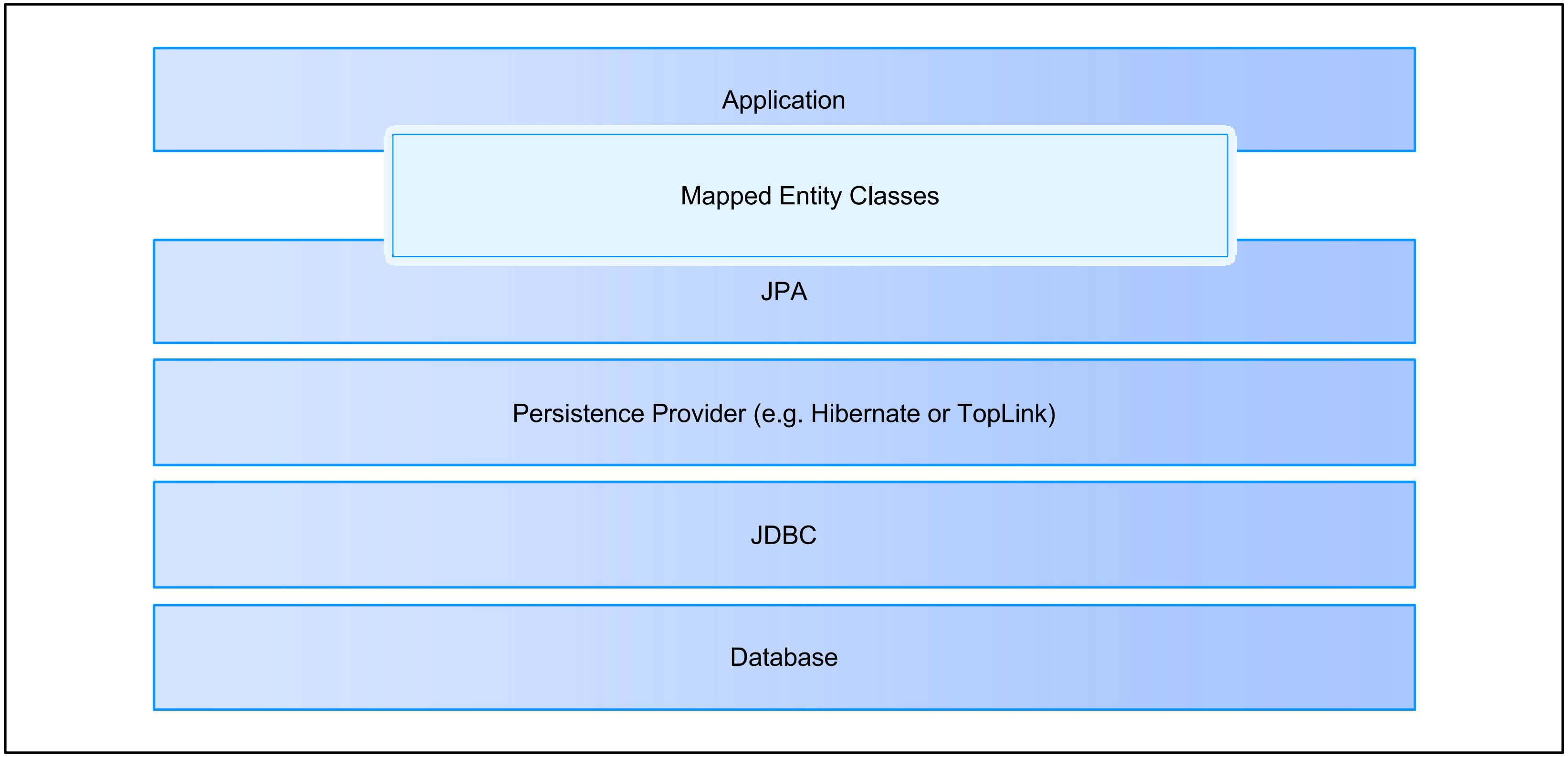
Using Spring with JPA

Spring Data Overview

Using Spring Data

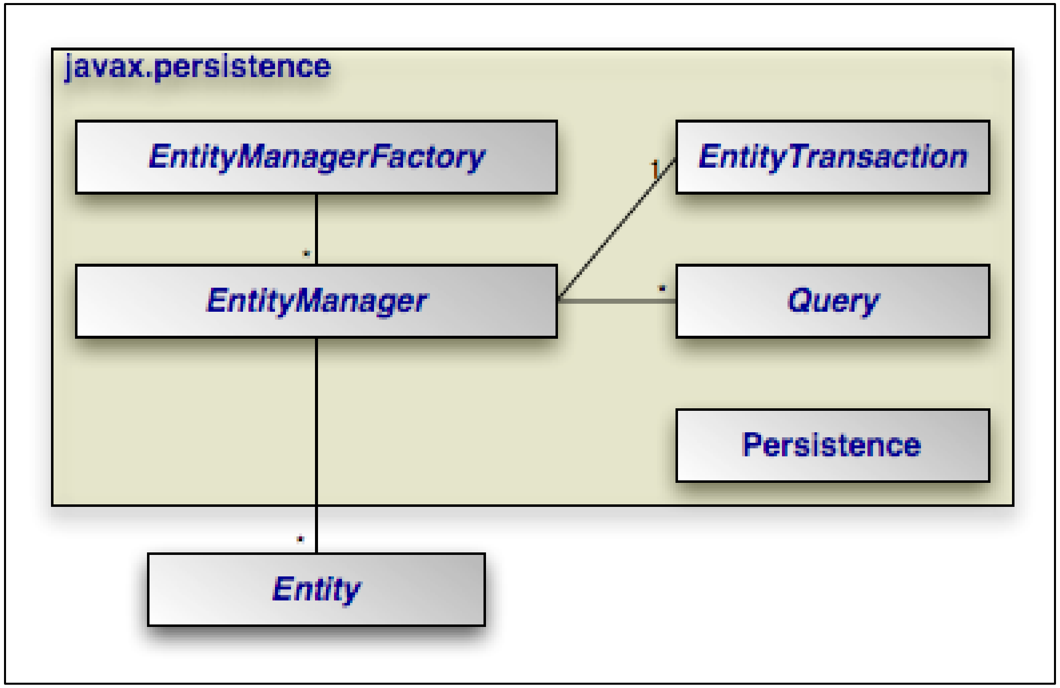
JPA: Overview and Architecture

- **JPA**: **Java Persistence API** : Standard ORM mapping spec
  - **Annotation-based mapping**: Entity metadata describes DB mapping - JPA runtime provides persistence services / entities
  - Usually built on existing persistence provider (e.g. Hibernate)



JPA: Architecture – Key Types

- **Persistence**: For configuration of the system
- **EntityManager**: Manages persistent entities and provides persistence operations to clients
- **EntityManagerFactory**: Factory for EntityManagers
- **Query**: For finding entities
- **Entity**: POJO class mapped using JPA



JPA: Entity Class

- Entity: A **lightweight persistent domain object**
  - Represents data stored in a DB
  - Annotations describe mapping to database - as shown below
  - It uses an generated id, default mappings, and explicit mappings

```
// Much detail omitted - package, constructors, …
import javax.persistence.*;

@Entity
@Table(name="Employees")           // Table name is Employees
public class Employee {
   @Id   // ID property - with generated id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   @Column(name="id")                // Column name is id
   private Long       id;
   private String     firstName;   // Uses defaults
   private String     lastName;    // Uses default
   @Column(name="compenation")     // Column name is compensation
   private BigDecimal salary;
}
```

JPA: Persistence Unit Configuration

- **Persistence unit**: Defines the set of entities managed by an entity manager, plus DB info and other info
  - Generally configured in XML (*persistence.xml*), as shown below

```
<persistence xmlns="…" version"2.0"> <!-- namespaces not shown -->
   <persistence-unit name="javatunesPersonnel"
                     transaction-type="RESOURCE_LOCAL">
      <properties>
        <property name="hibernate.connection.url"
                  value="jdbc:derby://localhost:1527/PERSONNEL"/>
        <!-- Other DB properties omitted -->
       <property name="hibernate.dialect"
             value="org.hibernate.dialect.DerbyTenSevenDialect"/>
      </properties>
   </persistence-unit>
</persistence>
```

JPA: EntityManager

- **EntityManager** (or **EM**): Used to interact with a database, and to do CRUD operations on an entity
  - Configured in *persistence.xml*
  - Generally, inject into JEE components (e.g. an EJB, servlet, etc.)
    - Can inject the same way into Spring components as we'll see soon
  - Once you have an EM, persistence operations are easy
    - e.g. - the **`em.find()`** call below is all that's needed to get an `Employee`
    - We ignore TX control in this example

```
import javax.persistence.PersistenceContext; // Other imports omitted

@Stateless
public class EmployeeServiceImpl implements EmployeeServiceLocal {
  @PersistenceContext // Inject - Spring uses this JPA annotation also
  private EntityManager em;

  public Employee searchById(Long id) {
    return  em.find(Employee.class,id);
  }
}
```

Spring Support for JPA

- **JPA**: **Java Persistence API**
  - Standard Java ORM mapping specification
  - Uses annotations for mapping
  - An `EntityManager` executes persistent operations

- Spring supports **JPA** similarly to how it supports Hibernate
  - Providing classes to create JPA based repositories
  - In package `org.springframework.orm.jpa`

- Spring provides classes for managing/injecting entity manager factories and entity managers

Managing the EntityManager[Factory]

- Spring supports management and injection of an `EntityManager` (**EM**) and/or `EntityManagerFactory` (**EMF**)

- Multiple ways to accomplish this
  – JNDI lookup, or the classes below (in `org.springframework.orm.jpa`)

1. **JNDI lookup**: Look up EMF from JEE environment using Spring's JNDI capabilities
   • Configuration done on JEE side

2. **`LocalContainerEntityManagerFactoryBean`**: Creates a container managed EMF
   • Flexible configuration

3. **`LocalEntityManagerFactoryBean`**: Creates an EMF suitable for integration testing or standalone programs (only)
   • Least flexible in terms of configuration

1. JEE: Obtaining an EMF From JNDI

- Use Spring JNDI to lookup EMF when using JEE:
  - Depends on JEE container for JPA setup
  - JNDI name also configured in JEE container
- Persistence unit deployment done by the JEE container
  - As is `DataSource` config and often transaction control
  - Spring just looks up the EMF, and you inject an EM into client code
    - Below, we look(via XML and `@Configuration`) up a persistence unit with JNDI name `persistence/personnelPU` (1)

```
<beans>
  <jee:jndi-lookup id="personnelEmf"
             jndi-name="java:comp/env/persistence/personnelPU/"/>
</beans>
```

```
@Bean
public EntityManagerFactory personnelEmf() throws NamingException {
return (EntityManagerFactory)
  (new InitialContext().lookup("java:comp/env/persistence/personnelPU/"));
}
```

2. LocalContainerEntityManagerFactoryBean

- Gives Spring-based apps full control over EMF configuration (1)
  - Appropriate for fine-grained customization
  - Supports datasource configuration and load-time weaving
  - Supports local and global transactions
  - Provides a **transaction-scoped container-managed** entity manager
  - May conflict with JEE server - with JEE, generally use JNDI lookup

- Generally you inject an EM into a JPA-based repository class
  - The EM is associated with, and scoped to, a transaction (TX)
  - The EM associated with the current TX is propagated with the TX
  - When the TX finalizes, the EM is flushed/closed
  - This is the default behavior – you can specify differently (2)

- Uses **standard JPA annotations** to inject the EM
  - We'll illustrate later

Spring/JPA Integration Configuration

- Additional configuration required for Spring/JPA integration
- **Annotation Scanning**
  - JPA annotations for injecting the EM must be detected (1)
  - `AnnotationConfigApplicationContext` includes this support
  - The following XML element also includes this support:

```
<context:annotation-config/>
```

- **Vendor Adaptor**: Configures Spring/JPA-provider integration
  - Supports common config properties, including:
    - Should SQL traces be logged
    - The database platform
    - Should the schema be generated in the DB when the app starts
- See example on next slide (Hibernate-based)

Example: XML Configuration

- For LocalContainerEntityManagerFactoryBean
  – See next slide for @Configuration example

```
<beans …> <!-- namespaces not shown -->

  <context:annotation-config/>

  <bean id="vendorAdapter" class=
      "org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <property name="databasePlatform"
              value="org.hibernate.dialect.DerbyTenSevenDialect"/>
    <property name="showSql" value="true"/>
    <property name="generateDdl" value="false"/>
  </bean>

  <bean id="personellEmf" class=
     "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="personnelDataSource"/>
    <property name="persistenceUnitName" value="javatunesPersonnel"/>
    <property name="jpaVendorAdapter" ref="vendorAdapter"/>
  </bean>
</beans>
```

Example: @Configuration

- We show an `@Configuration` equivalent

```
@Bean
public JpaVendorAdapter vendorAdapter() {
  JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
  vendorAdapter.setDatabasePlatform(
          "org.hibernate.dialect.DerbyTenSevenDialect");
  vendorAdapter.setShowSql(true);
  vendorAdapter.setGenerateDdl(false);
  return vendorAdapter;
}

@Bean
public LocalContainerEntityManagerFactoryBean personnelEmf() {
  LocalContainerEntityManagerFactoryBean em =
                    new LocalContainerEntityManagerFactoryBean();
  em.setDataSource(dataSource());  // assume defined elsewhere
  em.setPersistenceUnitName("javatunesPersonnel");
  em.setJpaVendorAdapter(vendorAdapter());
  return em;
}
```

3. LocalEntityManagerFactoryBean

- Suitable for standalone / testing environments
  - JPA provider's autoscanning detects persistent classes
  - Generally requires the persistence unit name (in *persistence.xml*)
  - Very little configuration available for this bean
    - No support for using a Spring-configured datasource
    - No global transaction support
    - No support for specify a JVM agent for load-time weaving
  - Below is sample config (persistence unit name is "`personnel`")

```
<bean id="personnelEmf"
    class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="personnel"/>
</bean>
```

```
@Bean(name="entityManagerFactory")
public LocalEntityManagerFactoryBean javatunesEmf() {
  LocalEntityManagerFactoryBean emf = new LocalEntityManagerFactoryBean();
  emf.setPersistenceUnitName("javatunesPersonnel");
  return emf;
}
```

JPA Repository / DAO

- A JPA-based Repository is shown below
  - Spring supports **@PersistenceContext**, to inject the EM (1)
    - Standard JEE annotation in `javax.persistence`
    - Recognized via post-processors from `context:annotation-config`
    - Usable with all the EMF configuration choices we discussed
  - If an active TX/EM is present, it's injected
    - Otherwise a new EM is created and injected
- That's it – no Spring code, no need to manually propagate the EM
  - We'll see how useful this is shortly

```
public class JpaEmployeeRepository implements EmployeeRepository {
  @PersistenceContext
  private EntityManager em;  //get/set methods not shown

  public Employee searchById(Long id) {
    return  em.find(Employee.class,id);
  }
}
```

Extended Persistence Context

- Used for an EM/persistence context that lives longer than a TX
  - In that case, you must create the EM yourself
  - Can inject an entity manager factory (using the JPA annotation **@PersistenceUnit**), and create the EM with it
  - You control the EM lifecycle in your code
    - e.g. closing it when you're done (see notes)

- The code fragment at bottom shows creation of the EM
  - You would close it when you're done with it

```
public class JpaEmployeeRepository implements EmployeeRepository {
  @PersistenceUnit
  private EntityManagerFactory emf;  //get/set methods not shown
  private EntityManager em;

  public JpaEmployeeRepository {
    em = emf.createEntityManager();
  }
}
```

[Optional] Lab 5.1: Integrating Spring and JPA

In this lab, we will work with the Spring/JPA integration

Spring Data Overview

ORM Overview

Using Spring with Hibernate

Using Spring with JPA

Spring Data Overview

Using Spring Data

Why do We Need Spring Data

- ORM frameworks improve data access dramatically
  - **Automatic assembly** of retrieved objects from relational data
  - **Inheritance and polymorphism**[1]
  - **Improved performance through caching** of queried objects

- Still a lot of tedious and repetitive work
  - Basic persistence operations **are nearly identical**
    - CRUD - create, read, update, delete
  - **Even "custom" queries are very similar**

- Lots of ways to store and access data
  - Relational databases (Oracle, MySQL, etc), NoSQL (HBase …)
  - Many APIs - JDBC, JPA, proprietary APIs

Spring Data Goals

- Significantly **reduce the amount of boilerplate code**
  – In simple cases, no code at all is needed
  – Common operations/queries are **generated**
- Provide **easy customization** as needed
  – If it doesn't do what you need, it's **easy to add your own data access code**
  – **Still benefiting** from the places where Spring Data does work
- Provide a **common and consistent model** for data access
  – Easing support for multiple data stores / access methods
  – Making migration easier and faster

Spring Data Project

- **http://projects.spring.io/spring-data/**
- Provide a **consistent** Spring-based model for data access
  – While **supporting the special traits** of underlying data stores
  – Higher layer over an API like JPA
- Supports many data access technologies and data sources
  – JDBC, Hibernate, JPA, etc.
  – Relational DB, non-relational (e.g. NoSQL), Map-Reduce (e.g. Hadoop), cloud-based, etc.
- Spring Data is an **umbrella** project
  – **Subprojects** exist for specific databases / access technologies
  – Each subproject includes experts from its specific technology

Spring Data Modules

- Separate **modules** support each area of capability, including:
  - **Spring Data Commons**: Core capability used by other modules
  - **Spring Data JPA**: Support for JPA-based repositories
  - **Spring Data MongoDB**: Support for MongoDB-based repositories
  - **Spring Data KeyValue**: Support for Map-based repositories for key-value stores
  - **Spring Data REST**: Exports repositories as RESTful resources
  - Others, e.g. Redis, Gemfire, and Apache Solr support

POM for Using Spring Data JPA

- Below we show a Spring Boot-based POM for Spring Data JPA
  - Will bring in **`spring-data-jpa`**, and any other needed dependencies
  - It's the easiest way to use Spring Data
  - You can depend directly on spring-data-jpa, but it leads to a more complicated POM

```xml
<dependencies>
     <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
     </dependency>
</dependencies>
```

LEARNINGPATTERNS

Using Spring Data

ORM Overview
Using Spring with Hibernate
Using Spring with JPA
Spring Data Overview
Using Spring Data

How is Spring Data (JPA) Structured?

- It is a set of **interfaces** and **conventions**
  – Lets Spring Data generate data access code for you

- Three core **interfaces**

```
interface Repository<TID>
```

  - **Marker interface** defining the entity type (T) and the id type (ID)

```
interface CrudRepository<T,ID> extends
  Repository<T,ID>
```

  - Defines common methods requiring no implementation, e.g.

```
T findOne(ID primaryKey);
```

```
interface JpaRepository<T,ID>
```

  - JPA specific version of `CrudRepository`
  - In Spring Data JPA subproject
  - Adds a few methods, e.g. `getOne()`, and optimizes others for JPA

CrudRepository/JpaRepository Methods

- You get the `CrudRepository` methods below for free
  - package `org.springframework.data.repository`
  - **`long count()`**: Returns the number of entities available
  - **`void delete(ID id)`**: Deletes the entity with the given id
  - **`void deleteAll()`**: Deletes all entities managed by the repository
  - **`Iterable<T> findAll()`**: Returns all instances of the type
  - **`Optional<T> findById(ID id)`**: Retrieves a single entity by its id (1)
  - **`<S extends T> S save(S entity)`**: Saves a given entity
    - Many other methods - see the javadoc
- **`JpaRepository`** adds/modifies several methods, including
  - **`List<T> findAll()`**: Returns a list, not an Iterable
  - **`T getOne(ID id):`** Return entity with id, or null if not found
    - A few others - guess where to look
  - package `org.springframework.data.jpa.repository`

Mini-Lab: Review Javadoc

Mini-Lab

- We provide the Spring Data Commons Javadoc
  - Under *StudentWork/Spring/**Resources/SpringDataDocs/Commons/javadoc***
  - In a browser, open *index.html* in the folder above
- Review the javadoc for the following types
  - `CrudRepository`

- We provide the Spring Data JPA Javadoc
  - Under *StudentWork/Spring/**Resources/SpringDataDocs/JPA/javadoc***
  - In a browser, open *index.html* in the folder above
- Review the javadoc for the following types
  - `JpaRepository`

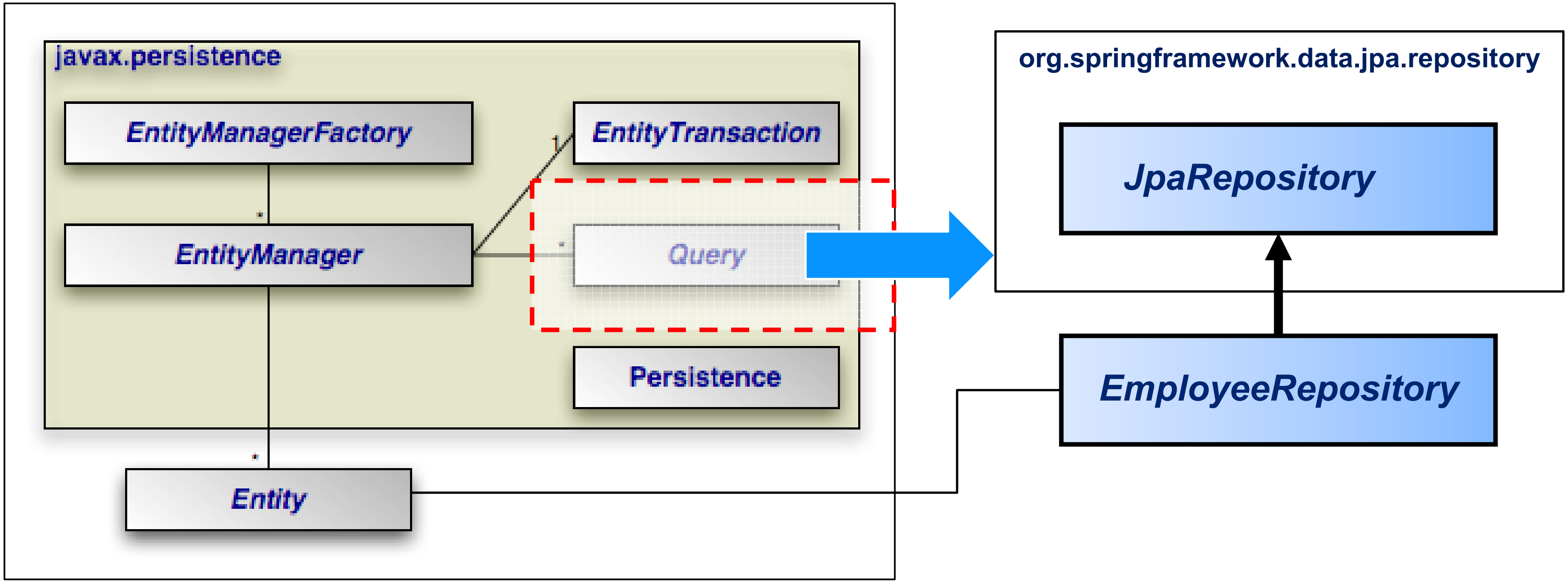
The Employee Repository Type

- Below, we define the interface for an `Employee` repository
  - Simply extend **`JpaRepository<Employee,Long>`**
  - Defines repository for our `Employee` entity with id type of `Long`
- To enable Spring Data, we configure it as shown at bottom
  - **`@EnableJpaRepositories`** enables a JPA-based repository
  - We specify the base packages to scan, and the name of the JPA `EntityManagerFactory` bean
  - See notes for XML version

```
package com.javatunes.persistence;  // Some detail omitted …
import org.springframework.data.jpa.repository.JpaRepository;
public interface EmployeeRepository extends JpaRepository<Employee,Long> {}
```

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
@Configuration
@EnableJpaRepositories(basePackages = {"com.javatunes.persistence"},
                       entityManagerFactoryRef="javatunesEmf")
public class SpringRepositoryConfig { /* Other detail omitted … */ }
```

Structure of our Repository



- `JpaRepository` is a Spring Data interface
  - Provides basic CRUD methods optimized for JPA
- `EmployeeRepository` interface extends `JpaRepository`
  - Implementation class is **generated by Spring**
  - **EmployeeRepository is a template for Employee queries**

Using the Repository

- Below, we show simple code that uses the repository
  - It's injected into the class (standard Spring DI)
  - We use it to get an employee by id
- What's different? We **didn't write any implementation**!
  - We **didn't even define** a method
    - `getOne()` is inherited from `JpaRepository`
  - We **didn't define** a repository implementation
    - It was generated by the Spring Data JPA framework

```
public class UseEmployee {

   @Autowired
   EmployeeRepository repo;

   public void findAnEmployee() {
      Employee found = repo.getOne(1L);
   }

}
```

Using Other CrudRepository Methods

- Below, we use some of the other methods
  - These methods are inherited from **CrudRepository**
  - Implementations are generated by the framework

```
public class UseEmployee {

   @Autowired
   EmployeeRepository repo;

   public void workWithEmployee() {
      for (Employee cur : repo.findAll()) {
         System.out.println(cur);
      }
      System.out.format("Repo count is %d\n", repo.count());

      Employee found = repo.getOne(1L);
      repo.delete(found);
      System.out.format("Count after delete is %d\n", repo.count());
   }

}
```

Lab 5.2: Using Spring Data

In this lab, we will work with Spring Data to create a Spring-Data-JPA repository interface for MusicItem

Defining Queries Using Naming Conventions

- **Naming conventions** support defining new (auto-generated) query methods in your interface
  - Easily extend query functionality based on your types
- e.g. - Employee has a **`firstName`** property
  - Below, we define methods to find, count, and remove employees based on first name
  - The implementations are generated automatically

```
// Or extend Repository if you don't want CRUD methods (1)
public interface EmployeeRepository extends
                              JpaRepository<Employee,Long> {

  List<Employee> findByFirstName(String firstName);

  Long countByFirstName(String firstName);

  Long deleteByFirstName(String firstName);
 }
```

More About Generated Queries

- Below, we define two new queries in our repository interface
  - **`findByFirstName()`** queries by an exact match on `firstName`
  - **`findByFirstNameIgnoreCase()`** ignores case in the match
- The framework uses the name of the method to generate the corresponding implementation
  - a **`findByXXX()`** method is a query using the value of property **XXX**
  - Again - no implementation code is required

```
public interface EmployeeRepository extends CrudRepository<Employee,Long> {
   public List<Employee> findByFirstName(String firstName);
   public List<Employee> findByFirstNameIgnoreCase(String firstName);
}
```

```
  // Rest of detail omitted - as in previous examples
  public void workWithEmployees() {
     List<Employee> allJanes  =
                 repo.findByFirstName("Jane"); // Finds by first name Jane
     allJanes = repo.findByFirstNameIgnoreCase("jane");  // Find jane also
  }
```

More Complex Queries

- You can combine property expressions with **And** and **Or**
- You can use operators like **Between**, **LessThan**, **GreaterThan**, and **Like**
- We define a few in the example below
  - We use one of the in the example at bottom

```
public interface EmployeeRepository extends JpaRepository<Employee,Long> {
  // Find with specific salary with given first name
   public List<Employee> findBySalaryAndFirstNameIgnoreCase(
                                          BigDecimal salary, String firstName);

  // Find with salary greater than given, and then with salary between values
   public List<Employee> findBySalaryGreaterThan(BigDecimal salary);
   public List<Employee> findBySalaryBetween(BigDecimal low, BigDecimal high);
}
```

```
  // Rest of detail omitted - as in previous examples
  BigDecimal low, high; // Initialized in some manner
  List<Employee> employees= repo.findBySalaryBetween(low, high);
```

Configuring Results

- You can use **OrderBy** to order the results
  - In the example below, we order by `lastName`
- You can limit the number of results via **First** or **Top**
  - In the example below, we get the first 2 employees
  - Note: A **query** method is the same as a **find** method

```
public interface EmployeeRepository extends JpaRepository<Employee,Long> {
  // Find with greater salary, order by last name
  public List<Employee>
    findBySalaryGreaterThanOrderByLastName(BigDecimal salary);

  // Get first 2 employees with greater salary, order by last name
  List<Employee>
     queryFirst2BySalaryGreaterThanOrderByLastName(BigDecimal salary);
}
```

```
  // Find first 2 by minimum salary, order by last name
  BigDecimal salary; // Initialized in some manner
  List<Employee> employees=
     repo.queryFirst2BySalaryGreaterThanOrderByLastName(salary);
```

Defining Queries with JPQL

- You can define queries using the JPA Query Language
  - Include **@Query** annotation on interface method
    - Method can have any name
  - Provide JPQL query string (1)
  - We illustrate below

```
// Or extend Repository if you don't want CRUD methods (1)
public interface EmployeeRepository extends
                      JpaRepository<Employee,Long> {

  // Query using positional parameters
  @Query("select e from Employee e where e.firstName like %?1")
  List<Employee> findByFirstnameEndsWith(String firstname);

  // Same query using named parameters
  @Query("select e from Employee e where e.firstName like %:firstName")
  List<Employee> findByFirstnameEndsWith2(
                      @Param("firstName")String firstName);

}
```

Lab 5.3: Writing Query Methods

In this lab, we create queries in our interface using the Spring Data Naming Conventions

Review Questions

- What is a Repository class / DAO, and how is it used?
- How does Spring support Hibernate?
- How does Spring support JPA?
- How does Spring Data make database access easier?

Lesson Summary

- A **Repository/DAO** encapsulates DB access
  - Insulates your app code from dealing with database details
  - Gathers all DB code in one place, for easy management
- **Spring** provides extensive support for creating repository beans
  - If using JDBC directly, it can manage most of the low-level details
  - e.g., acquiring and releasing connections
- Extensive support for ORM technologies like **JPA** and **Hibernate**
  - **Hibernate**: Supports configuration of a `SessionFactory`
    - Easy integration with our Spring beans, e.g. datasources
    - Depends on contextual sessions for session propagation
  - **JPA**: Supports injection of entity managers or entity manager factories
    - Support for configuring the entity manager factory
    - Easy integration of Spring and JPA, with transparent Spring support
    - An entity manager can be injected with standard Java annotations

Lesson Summary

- **Spring Data** is a set of **interfaces** and **conventions**
  - Organized in modules for different data access methods like JPA
- You can write a repository by using one of the interfaces
  - You often don't need to write implementation code
  - As we saw for our JPA-based `EmployeeRepository`
  - You enable/configure the repositories in Spring
    - As we saw with **`@EnableJpaRepositories,`**
    - You then just inject and use the repository
  - It's transparent to the client that Spring Data was used
- Repositories have many capabilities
  - e.g. complex queries using multiple predicates and qualifiers
  - None of them require **ANY** code be written

Session 6: Transactions

Spring Transaction Management

@Transactional Configuration

Pointcut-based Configuration

Lesson Objectives

- Understand and configure Spring transaction managers
- Understand how declarative transaction management works
- Use Spring declarative transaction management with both annotations and Pointcut-based Configuration

Spring Transaction Management

Spring Transaction Management

@Transactional Configuration

Pointcut-based Configuration

General Transaction (TX) Overview

- **Transaction**: A collection of actions on the state of a system
  - Transformation of a system from one consistent state to another
  - The collection of actions must conform to the **ACID** properties below

- **Atomicity**: All a transaction's work is saved or none is saved
  - In a bank transfer both debit and credit must occur (1)
- **Consistency**: A TX is a correct transformation of the state
  - In a bank transfer the credit and debit are the same amount
- **Isolation**: Concurrent transactions appear to occur sequentially
  - A bank transfer can ignore other transactions on the accounts
- **Durability**: Once a transaction completes successfully (commits), its changes to the state survive failures – like a server crash

General Transaction Lifecycle

- A transaction is begun in a system [(1)]
  - – Subsequent operations are part of the TX
  - – Associated operations in other programs (e.g. a DB) join the TX
- **Committing** declares the operations to be complete & correct
  - – Once the transaction commits, its effects are durable
- **Rolling back** undoes the operations
  - – The system can cause a rollback if it detects some failure
- Transactions support modularization of a system
  - – Each module is a transaction (or sub transaction)
  - – On success, the TX commits - all changes are made durable
  - – If something fails, the transaction is rolled back - no changes are saved

Spring's Transaction Managers

- **Transaction managers** abstract the underlying resource controlling transactions
  - Code isn't coupled to a transaction implementation
  - The transaction manager delegates to the underlying resource
    - They do NOT handle the transaction themselves

- Spring supplies several transaction managers
  - Supporting direct JDBC, Hibernate, JEE, JPA, etc.
  - Supports, but does **not require** JTA (JEE TX standard)
    - Apps can run under Java SE

- Each transaction manager requires appropriate resources
  - Generally configured in the Spring config
  - e.g., the JDBC transaction manager requires a `DataSource`,

Configuring Transaction Managers

- Below, are two configuration examples (XML and Java config)
  - **HibernateTransactionManager** for a Hibernate `SessionFactory`
  - **JpaTransactionManager** for a JPA `EntityManagerFactory`
  - See notes for JPA config using XML

```
<bean id="personnelSessionFactory" <!-- Hibernate -->
  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <!-- Detail not shown --> </bean>

<bean id="transactionManager" class= <!-- Hibernate TX manager -->
   "org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="personnelSessionFactory"/>
</bean>
```

```
@Bean // JPA transaction manager
public PlatformTransactionManager transactionManager(){
  JpaTransactionManager transactionManager =
               new JpaTransactionManager();
  transactionManager.setEntityManagerFactory(
    entityManagerFactory().getObject() ); // EMF declaration not shown
  return transactionManager;
}
```

Spring's JTA Transaction Manager

- **JtaTransactionManager** integrates Spring with JTA
  - JTA (Java Transaction API) support required in all JEE servers
- We show `JtaTransactionManager` configuration below
  - Autodetects the server's TX resources for most servers (1)
- Generally, generic `JtaTransactionManager` works
  - Sometimes need a server-specific version (Weblogic/WebSphere)
  - Can auto-configure the correct TX manager including for Weblogic and WebSphere by using **`<tx:jta-transaction-manager />`** (2)

```
<bean id="transactionManager"
  class="org.springframework.transaction.jta.JtaTransactionManager">
</bean> <-- Configure generic JTA TX Manager -->
```

```
@Bean
public PlatformTransactionManger transactionManager() {
  return new JtaTranscationManager();
} // Configure generic JTA TX Manager
```

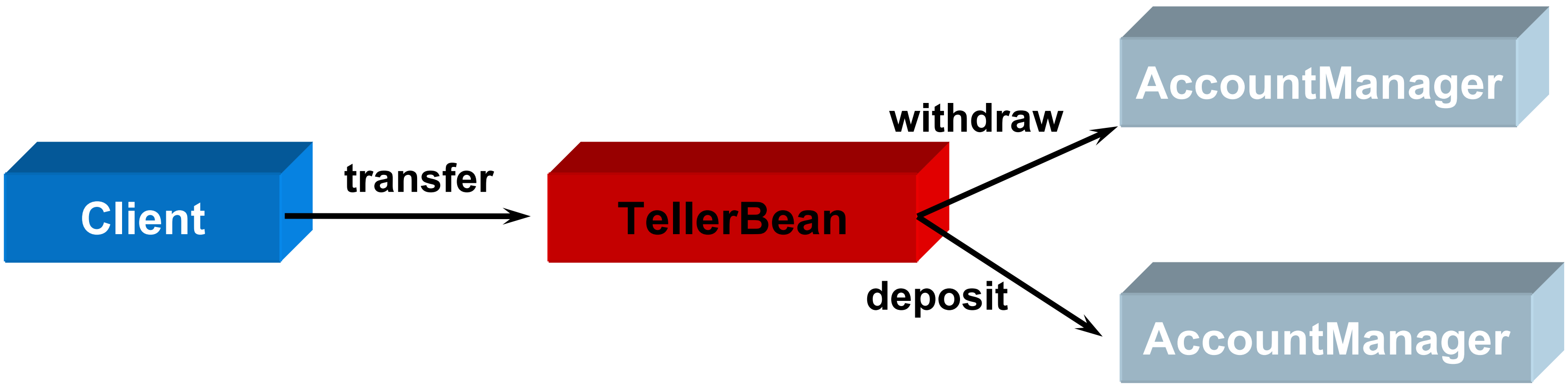
```
<tx:jta-transaction-manager /> <!-- Spring figures out what to use -->
```

Spring Declarative TX Management

- **Declares** a bean's transactional behavior
  - Rather than explicit coding (e.g. start/commit) via a TX API
  - Defines TX **boundaries** across which a TX **propagates**
    - Transactions are affected when crossing these boundaries
  - We'll cover the behavior first, then look at coding it
- Based on **Spring AOP**(1)
  - Spring intercepts the method calls and adds in TX behavior
  - Defaults to proxy-based interception
- TX declarations often done with annotations
  - Discussed later

Spring Transactional Scope

- Transaction **scope** - all the beans/resources in a TX
  - Transactions **propagate** as a bean invokes on other beans and to resources used from a bean (e.g. a datasource/DB)
  - Below, a TX in `TellerBean`, propagates to `AccountManagers`
  - If one of these accessed a DB, the TX would propagate to it also



Transaction Propagation

- An invocation's transaction context is determined by:
  - **Transaction attributes** on the invoked bean: e.g. REQUIRED
  - **The current transactional state**: Is a TX active?
  - Additional TX attributes for **isolation**, **timeout** of transactions, and for **read-only** transactions

- Depending on the above, the container may
  - **Start a TX**, if one is not active
  - **Propagate an existing TX** from one bean invocation to another
  - **Suspend** an existing transaction (to prevent its propagation)
  - This ensures all beans and resource participate appropriately in transactions

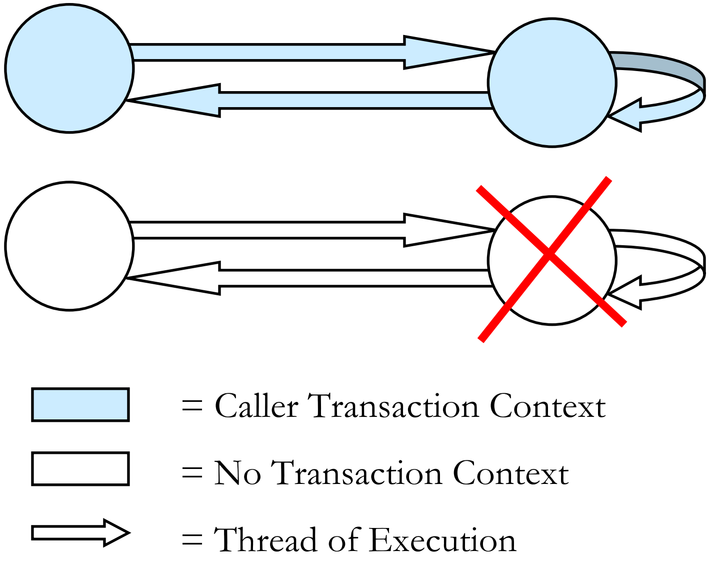
Transaction Attributes for Propagation

- Spring's transaction attributes controlling TX propagation
  - `MANDATORY`
  - `NESTED`
  - `NEVER`
  - `NOT_SUPPORTED`
  - `REQUIRED`
  - `REQUIRES_NEW`
  - `SUPPORTS`

- Usable at both a bean & method level
  - **Bean level**: The default for all methods in a bean
  - **Method level**: Behavior for the specific method
    - Overrides any bean level declaration

MANDATORY

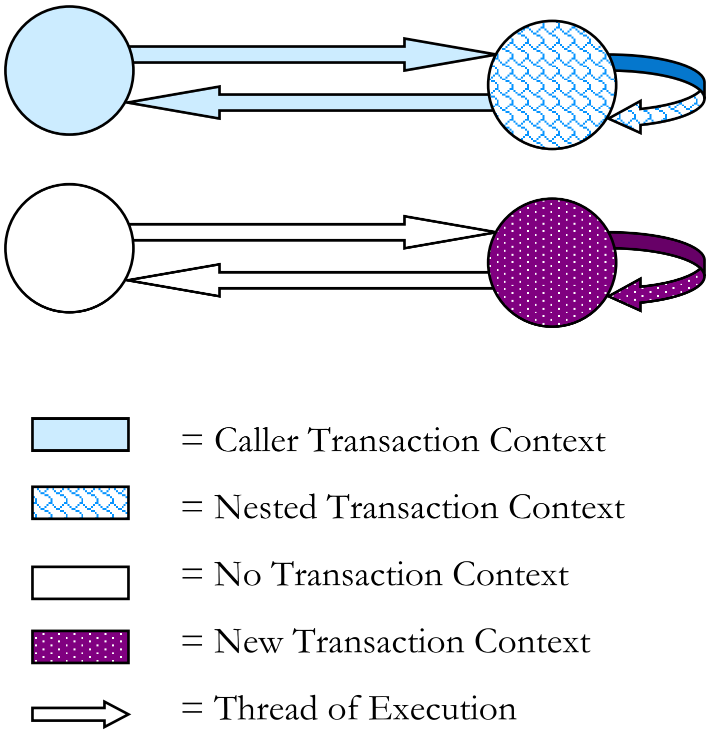
- Invocation **must** occur within the scope of a caller's TX
  - Invocation **in** a TX : TX is **propagated**
  - Invocation **not** in a TX : **`TransactionRequired` exception** thrown



- a) Incoming TX : TX Propagated
- b) No incoming TX : ERROR - exception thrown !

NESTED

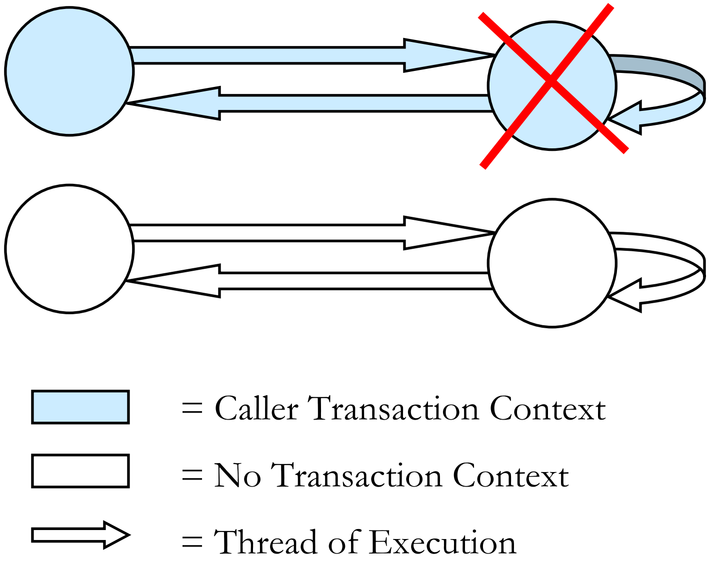
- Invocation always occurs in a TX context
  - Invocation **in** a TX : A **nested TX** is started
    - Container finalizes nested TX when its invocation completes
  - Invocation **not** in a TX : A **new TX** is started



- a) Incoming TX : Nested TX
- b) No incoming TX : New TX

NEVER

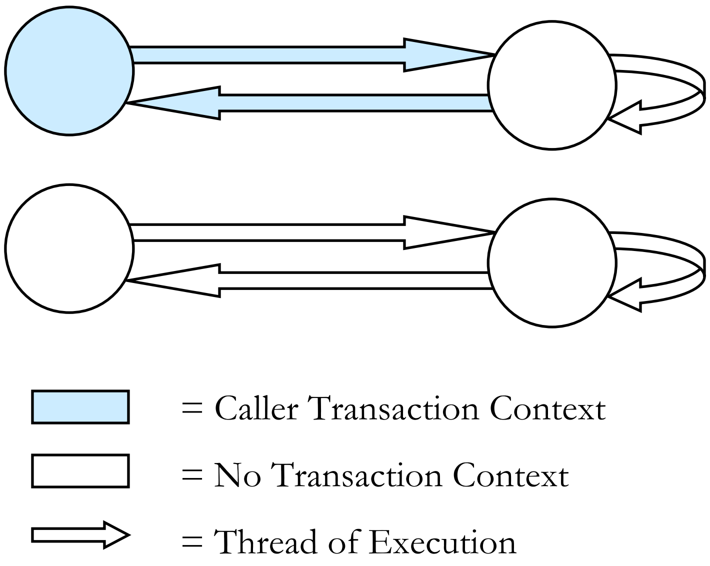
- Invocation **required** to occur **outside** the scope of a TX
  - Invocation **in** a TX : **ERROR** - Exception thrown
  - Invocation **not** in a TX : Invocation **without a TX** context



- a) Incoming TX : ERROR - exception thrown
- b) No incoming TX : No TX

NOT_SUPPORTED

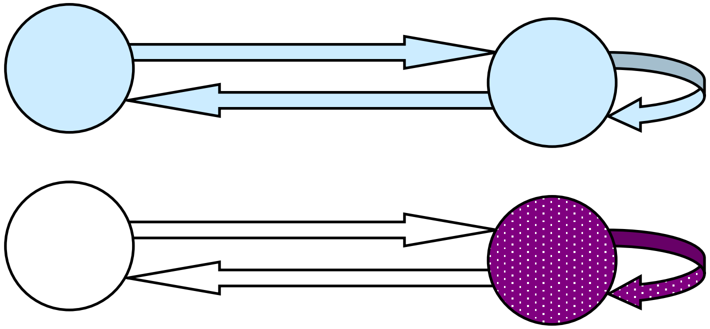
- Invocation should **not** be executed within a TX
  - Invocation **in** a TX : TX suspended, method executed, TX resumed
  - Invocation **not** in a TX : Invocation **without a TX** context



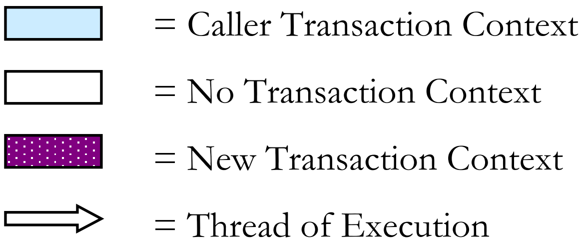
- a) Incoming TX : Suspended TX
- b) No incoming TX : No TX

REQUIRED

- Invocation always occurs in a TX context
  - Invocation **in** a TX : TX is **propagated**
  - Invocation **not** in a TX : A **new TX** is started
    - Container finalizes new TX when the invocation completes

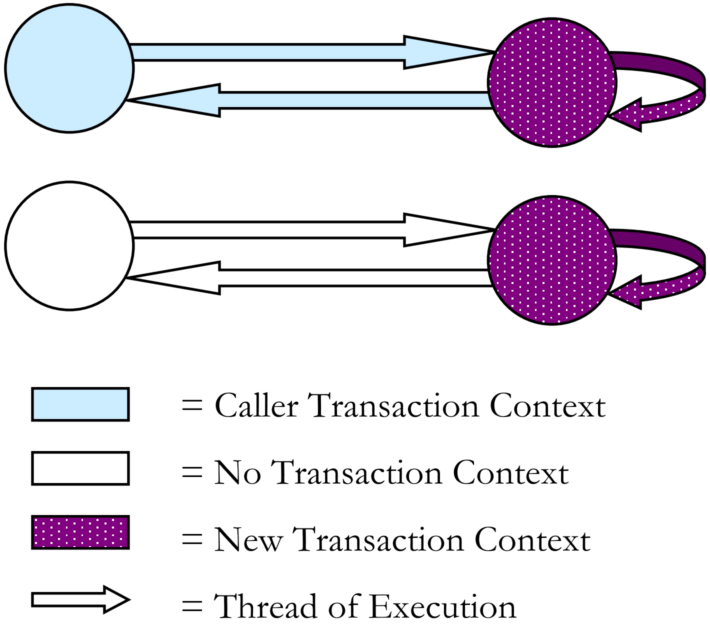


- a) Incoming TX : Propagated TX
- b) No incoming TX : New TX



REQUIRES_NEW

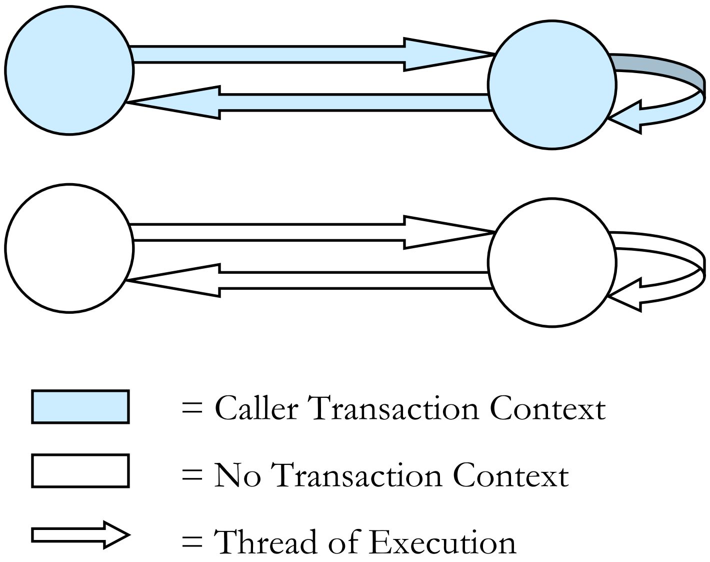
- Invocation always occurs in a **new TX context**
  - Invocation **in** a TX : **TX suspended**, **new TX** started
  - Invocation **not** in a TX : A **new TX** is started
    - In both cases, the new TX is finalized when the invocation completes



- a) Incoming TX : Suspended TX
- b) No incoming TX : New TX

SUPPORTS

- The bean method is invoked in the **caller's** TX scope
  - Invocation **in** a TX : TX is **propagated**
  - Invocation **not** in a TX : Invocation **without a TX** context



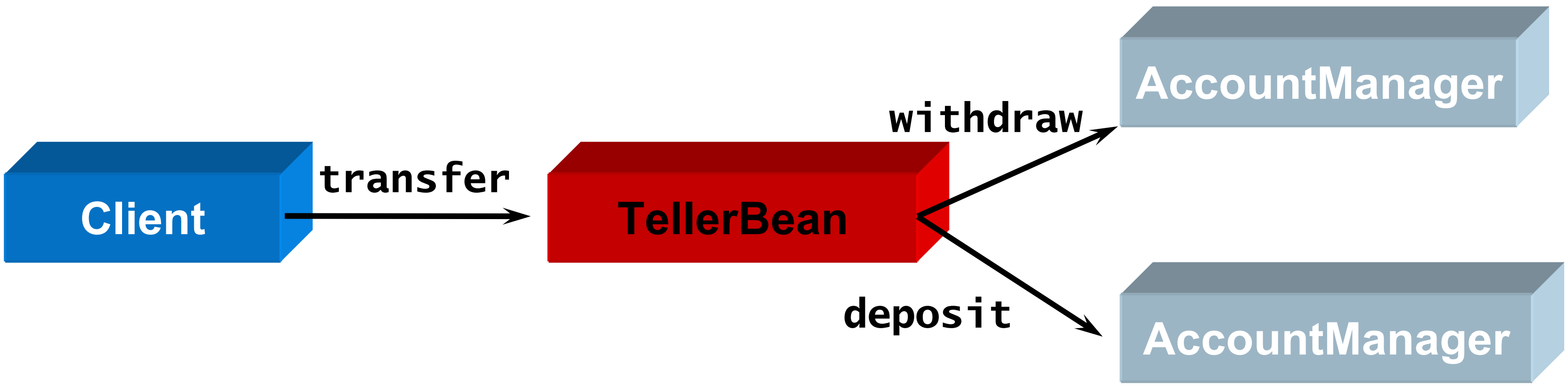
- a) Incoming TX : Propagated TX
- b) No incoming TX : No TX

Mini-Lab: Transaction Example

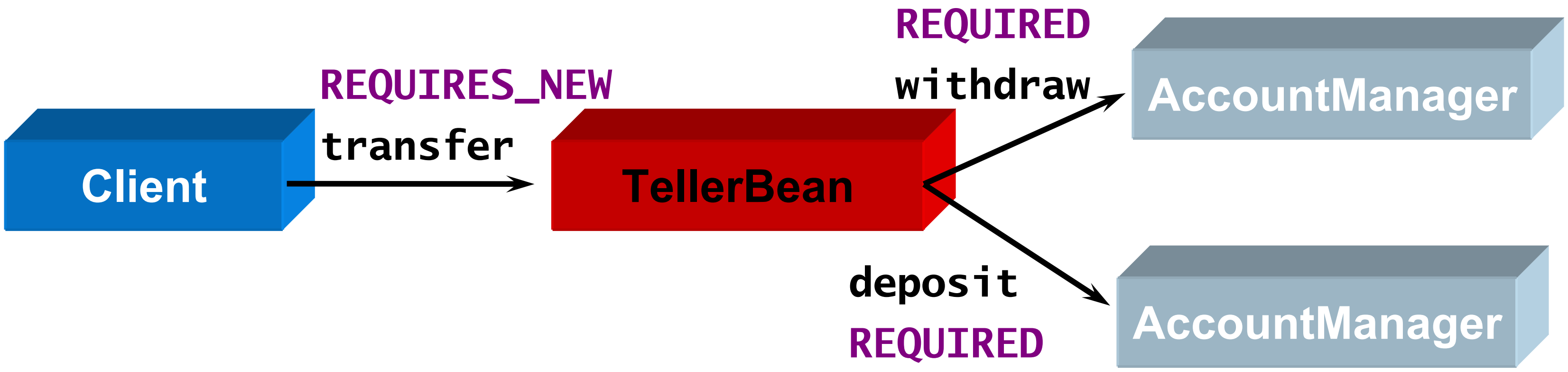
- A transfer involves two account objects - source and destination
  – How do you do this in one transaction?
- One way - encapsulate the transfer with another bean (e.g. a teller)
  – It withdraws from the source, and deposits to the destination

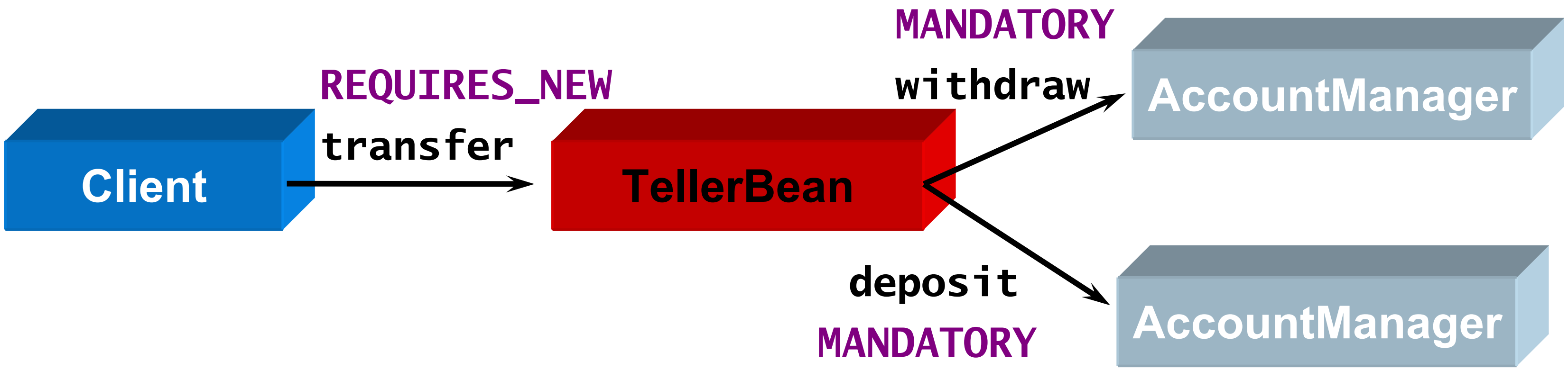
Mini-Lab

- In the following two slides discuss the TX behavior
  – What are the TX contexts at each TX boundary [(1)]

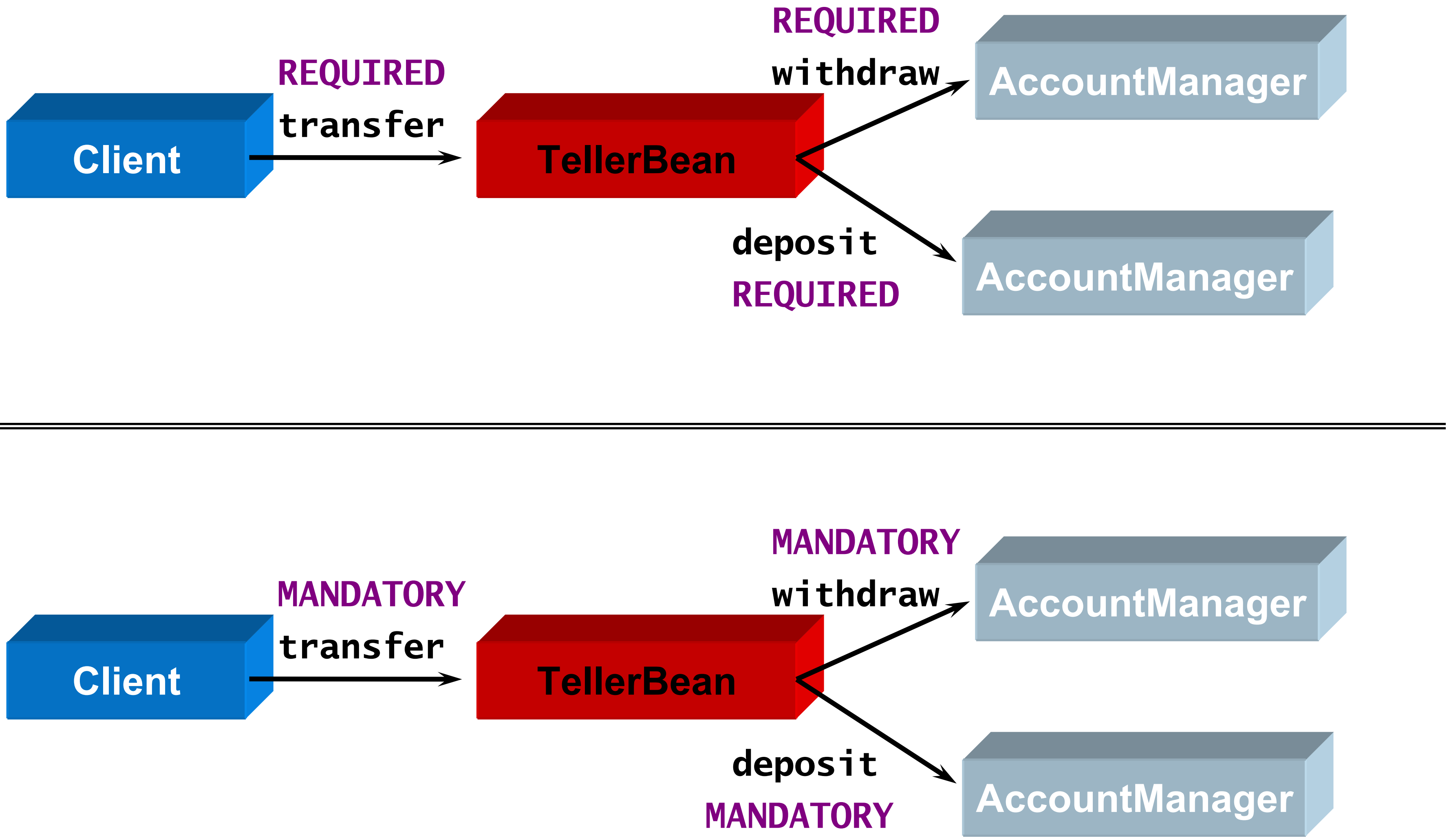


Transaction Attributes - Some Choices





Transaction Attributes - Some Choices



@Transactional Configuration

Spring Transaction Management

@Transactional Configuration

Pointcut-based Configuration

@Transactional Declarative Transactions

- Declarative transactions simplify programming
  - Simpler than a TX API
  - No TX code mixed with business code

- **`@Transactional`** can declare TX attributes
  - In package: `org.springframework.`**`transaction.annotation`**
  - Useable on a method or class
  - The **propagation** attribute declares the TX propagation
    - Values are members of the enum **`Propagation`** (1)
    - Default: **`REQUIRED`**

- `@Transactional` requires one of
  - **`<tx:annotation-driven>`** (XML config (2))
  - **`@EnableTransactionManagement`** (`@Configuration` config)

Additional Transactional Attributes

- Other attributes of `@Transactional` include:

- **`Isolation isolation`**: Degree of isolation this TX has from the work of other transactions
  - DEFAULT, READ_COMMITTED, READ_UNCOMMITTED, REPEATABLE_READ, SERIALIZABLE

- **`int timeout`**: how long (seconds) TX may run before timing out
  - Default: Timeout of underlying TX system

- **boolean readOnly**: Set to true if TX is read-only (Default `false`) (1)
  - A read-only TX does not modify any data
  - Can be useful optimization (such as when using Hibernate)
    - Especially when reading in large object sets

Example: Specifying Transaction Attributes

- Below, `JpaEmployeeRepository` has TX propagation of **`REQUIRED`**
  - With **`readOnly=true`** - so all methods will run read only
  - `create()` overrides with **`@Transactional`**
  - So it does NOT run read only
- This is easy to configure
  - May be hard to maintain - we'll see other ways

```
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@Transactional(readOnly=true)
public class JpaEmployeeRepository implements EmployeeRepository {

 public Employee getOne(long id) { /* Details not shown */ }
 public List<Employee findByFirstName(String firstName) { /* … */ }

 @Transactional
 public Employee save(Employee emp) { /* Details not shown */ }
}
```

Transactional Attributes Guidelines

- Generally, TX annotations are placed on a **concrete class**
  - Not on an interface, which may not work in all situations *
  - Only **public methods** work with the TX annotations
    - Assuming the proxy-based implementation (details covered later)
- Your annotated classes should **implement an interface**
  - This allows the use of JDK dynamic proxies (see notes)
  - This is fairly standard when using Spring anyway
- Proxy mode (the default) is limited
  - Self-invocations will not be wrapped in transactions, even if annotated
  - We'll cover this later

Rolling Back and Exceptions

- **Default behavior**: Mark a TX for rollback if an unchecked (e.g. runtime) exception is thrown
  - To trigger a rollback in your code, throw a runtime exception
  - The container catches unhandled exceptions, and marks the active TX for rollback
  - Checked exceptions don't trigger rollbacks
- **Refine** rollback behavior with the following `@Transactional` elements
  - **`Class[] noRollbackFor`**: Classes that should NOT trigger a rollback
  - **`String[] noRollbackForClassName`**: Names of classes that should NOT trigger a rollback
  - **`Class[] rollbackFor`**: Classes that SHOULD trigger a tx rollback
  - **`String[] rollbackForClassName`**: Names of classes that SHOULD trigger a rollback

Spring Data JPA and Transactions

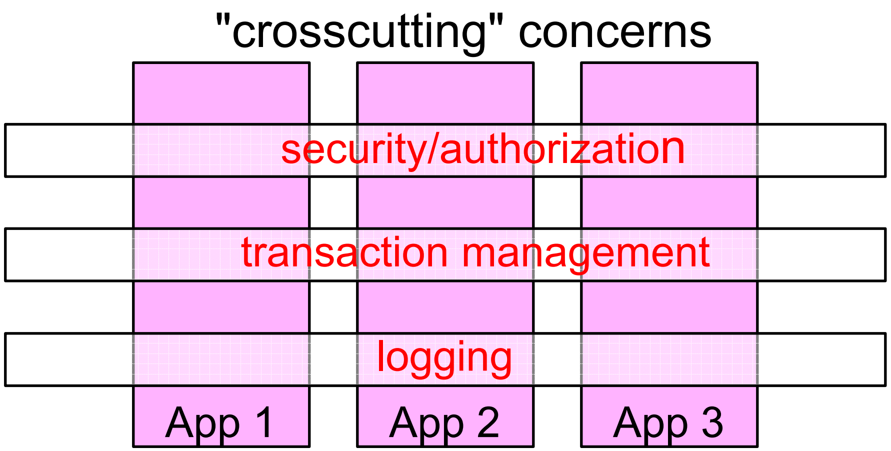
- Spring Data JPA sets TX behavior for all its methods
  - For **reading** operations, **`readOnly`** is set to **`true`**
  - All other operations configured with a plain **`@Transactional`**
    - So defaults apply
- Easy to set TX behavior
  - Just use **`@Transactional`** on your new methods (1)
  - **Redeclare** supplied method with your own TX settings

```
@Transactional(readOnly=true) // Sets defaults for all methods
public interface EmployeeRepository extends
                      JpaRepository<Employee,Long> {

  List<Employee> findByFirstName(String firstName); // Uses default TX

  @Transactional  // NOT read only
  Long deleteByFirstName(String firstName);

  @Override @Transactional(timeout = 10) // Override Spring Data TX
  public List<User> findAll();
 }
```

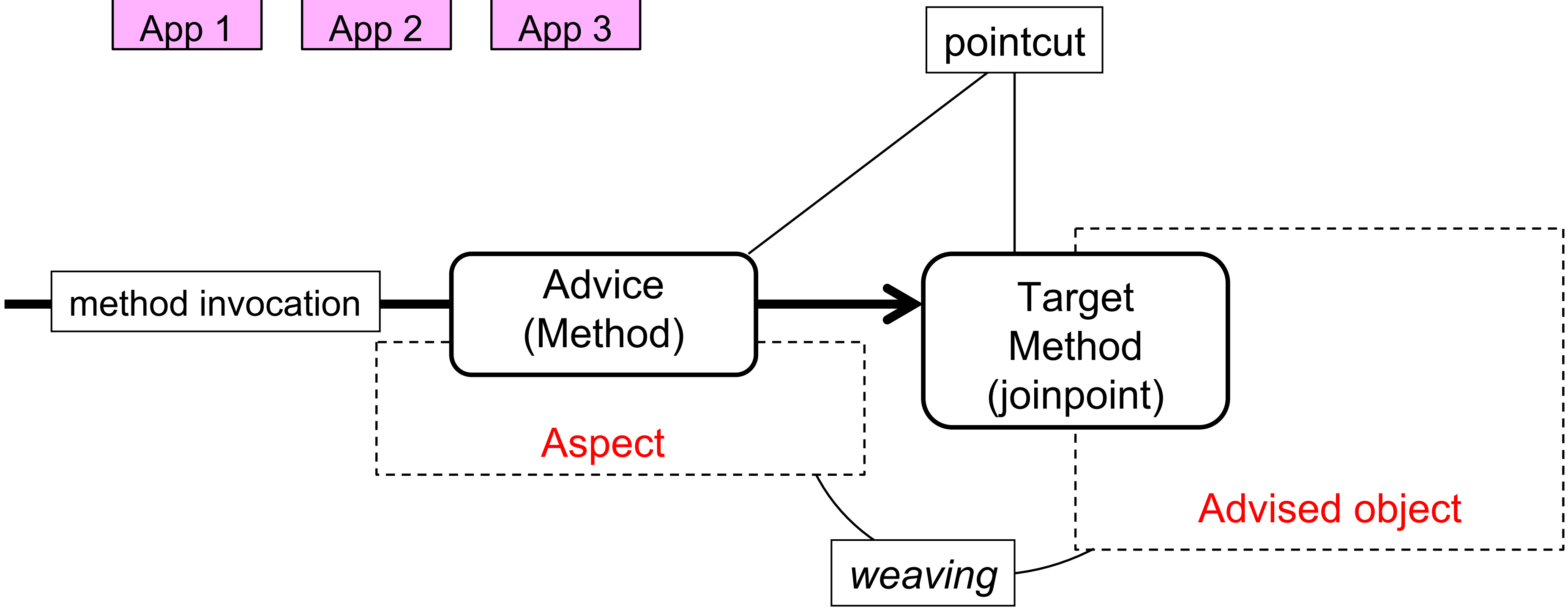
Aspect Oriented Programming (AOP) Defined

- Spring TX is built on top of AOP
- AOP **encapsulates crosscutting concerns** in **aspects**
  - **Separates** aspects from other system components
  - **Compose** aspects with other components to add their functionality
- Key AOP concepts
  - **Aspect**: A module holding cross-cutting concerns (a class in Spring)
  - **Advice**: The action taken by an aspect (its functionality)
  - **Joinpoint**: An execution point in a program, such as a method call, where advice may run
  - **Pointcut**: A specification matching joinpoints
    - Advice runs at any joinpoint that matches an associated pointcut
  - **Advised (target) object**: An object being advised by an aspect
  - **Weaving**: The combining of aspects with your application code
    - Creates advised objects where advice may run

Aspect Oriented Programming Illustrated

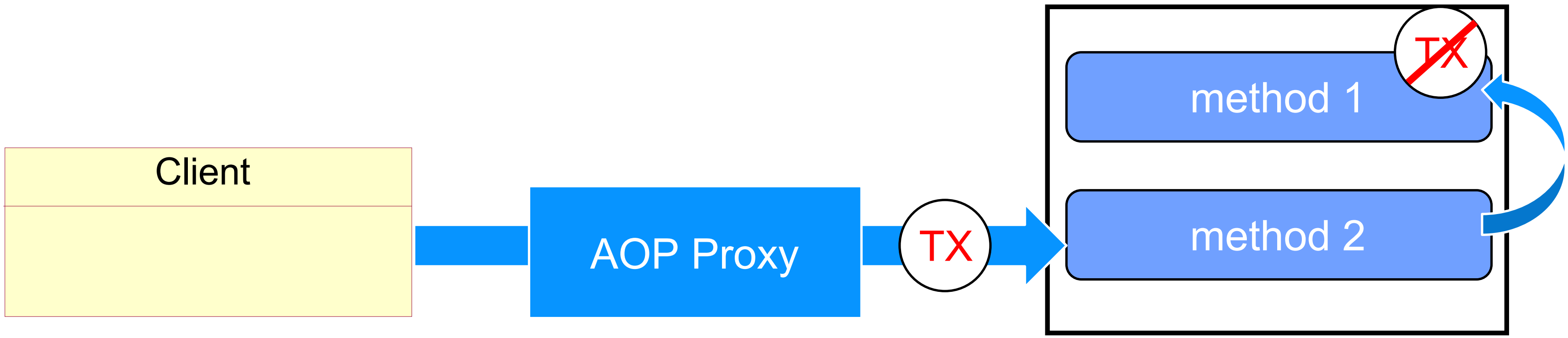


"Advice" is additional functionality added to existing functionality without directly changing it.



Spring TX and AOP

- Spring TX is based on Spring AOP
  - So all the characteristics of AOP apply to TX
- In particular, **self-invocation will NOT** trigger TX behavior
  - Unless load-time weaving is used
  - It's important to keep in mind with transactions
  - If you're not aware it's using proxy-based AOP under the hood, the behavior can be confusing



Example – Invoking Directly

- What happens when `empRep.createAll()` is called
  - `create()` called multiple times, and it's set to REQUIRES_NEW

```
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class HibernateEmployeeRepository
                                implements EmployeeRepository {

  @Transactional(propagation=Propagation.REQUIRED)
  public void createAll(Collection<Employee> employees) {
    for (Employee emp : employees) {
      create(emp);  // Call create() directly
    }
  }
  @Transactional(propagation=Propagation.REQUIRES_NEW)
  public int create(Employee emp) { /* Details not shown */ }
}
```

```
EmployeeRepository empRepo = ctx.getBean(EmployeeRepository.class);
Collection<Employee> emps = // Initialized in some manner …
empRepo.createAll(emps);  // What happens here
```

Example – Invoking Directly

- `create()` has a **`REQUIRES_NEW`** TX attribute in the example
  - But **that TX is NOT activated**
  - The call to `create()` goes through the **`this`** reference
  - It's **not a proxy**, so proxy-based behavior **doesn't run**
  - You could refactor your code to solve this
    - Cumbersome, not always possible
  - Another solution - configure Spring AOP so proxies not used
- Can configure Spring for **load-time weaving**, which doesn't have this issue, via:

  `<context:load-time-weaver/>`
  - Replaces the proxy-based behavior
  - Details are beyond the scope of the course

@Transactional Pros and Cons

Pros

- Transactional behavior in same class as executable code
- Simple to understand the annotations

Cons

- TX specification is scattered throughout your code
- Must specify TX behavior for each class individually
  - And for each method that doesn't follow the behavior of its class
- The above can lead to very hard to maintain code
  - Think of dozens or hundreds of transactional classes
  - Think about changing your transactional policy, and then having to revisit every single class and method

- We'll look at other ways to configure TX behavior

Lab 6.1: Spring Transactions

In this lab, we work with Spring transaction capabilities

[Optional] Pointcut-based Configuration

Spring Transaction Management

@Transactional Configuration

Pointcut-based Configuration

Spring Transactions and AOP

- Spring TX supports **pointcut-based** configuration
  - Uses AOP elements, and configurable separately from affected types

- We'll first cover some AOP basics
  - Only enough AOP to use it with transactions
  - Focusing on the parts useful for TX control (1)
  - There is more functionality available in AOP
    - Beyond the scope of this course

- AOP deals with **crosscutting concerns**
  - Transactions are the concern we'll focus on here
  - We'll first review the AOP elements we'll need

- We will cover XML configuration here
  - In this case, it's simpler and clearer than Java-based config

Defining a Pointcut

- **Joinpoint**: A location in your program
  - Where advice can run
  - In Spring, basically a **method** or all methods in a class/interface

- **Pointcut**: A specification matching joinpoints
  - It selects the joinpoints where advice can run
  - Specified via a **pointcut expression**
    - Specified with AspectJ's expression language
  - We illustrate on the next slide

- Pointcuts don't contain advice [1]
  - We'll soon link the pointcuts to TX behavior

Defining a Pointcut - XML

- Below we specify (via XML) a pointcut that matches "**any execution of a method in the Catalog type**" (1)
  - The pointcut is called **anyCatalogMethod**, (from the id value)
  - The AspectJ pointcut expression (more on this later) is this part:

execution(* com.javatunes.service.Catalog.*(..))

- To execute code with AOP, associate **advice** with a pointcut
  - It will run when that pointcut expression is matched
  - We'll see how this works with transactions soon

```
<aop:config>
  <aop:pointcut id="anyCatalogMethod"
     expression="execution(* com.javatunes.service.Catalog.*(..))"/>
</aop:config>
```

Specifying Transactions Using Pointcuts

- Spring TX's **pointcut-based** configuration uses AOP elements
  - Done via tags in **`tx`** / **`aop`** namespaces
  - We'll go briefly into some of the details here
  - The Java-based equivalent is very verbose

- On the next slide, is an example TX pointcut configuration (XML)
  - **`<tx:advice>`**: Defines the transactional semantics of methods
  - **`<tx:attributes>`**: Container for `<tx:method>` elements
  - **`<tx:method>`**: Specifies the TX attributes of the named method(s)
    - Method names can use wildcards, e.g.
    - **`find*`** means any method whose name starts with **`find`**

- We'll illustrate specifying TX behavior based on method names
  - One common convention to help organize TX behavior
  - Use the convention in your types, and you'll have appropriate TX semantics automatically injected

Example: Pointcut-based Transactions (XML)

- Below, we specify the following TX behavior
  - Methods with name `persistBatch` have **REQUIRED** TX propagation
  - Methods with names that **start with `find`** or **`persist`** have **REQUIRED** TX propagation (and **`find`** methods are **read only**)
  - The **`size`** method has **NEVER** TX propagation
- This advice defines the TX behavior
  - Next, we'll link it to a pointcut, which defines where it applies
  - Where the pointcut is matched, a TX is affected (e.g. started, stopped)

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="persistBatch" propagation="REQUIRED" />
    <tx:method name="persist*" propagation="REQUIRED" />
    <tx:method name="find*" propagation="REQUIRED"
               read-only="true" />
    <tx:method name="size" propagation="NEVER" />
  </tx:attributes>
</tx:advice>
```

Linking Advice With Pointcuts

- To run, TX advice is associated with a pointcut
- AOP pointcuts are used, as shown below (using XML config [1])
  - **`serviceOperations`** is a pointcut applying to any method of any type in the package `com.javatunes.service` (e.g. `Catalog`)
  - `<aop:advisor>` associates the pointcut to `txAdvice`
  - `CatalogImpl`'s **`findById`** and **`findByKeyword`** methods will now run with the `REQUIRED` TX attribute (and read only)
    - It's in package **`com.javatunes.service`**, and methods start with **`find`**

```
<aop:config>
  <aop:pointcut id="serviceOperations" expression=
              "execution(* com.javatunes.service.*.*(..))" />

  <aop:advisor pointcut-ref="jpaServiceOperations"
               advice-ref="txAdvice" />
</aop:config>
```

Resulting Behavioc

- The pointcut matches execution of any method on any class in `com.javatunes.service`
  - And triggers the advice (the defined TX behavior)
  - Execution of **`persistBatch`**, **`persist*`**, and **`find*`** methods in these classes run with the **`REQUIRED`** TX attribute (finds are read-only)
  - The **`size`** method runs with the **`NEVER`** TX attribute

- That's ALL you need to do to get the TX behavior
  - e.g. in the code below, the TX behavior is triggered

```
// Code fragment
Catalog cat;  // Initialized somehow

// The method below now runs with REQUIRED TX behavior
cat.findByKeyword("Sting");
```

<tx:method> Attributes

- `<tx:method>` includes the following attributes:
  - **`name`**: Method name pattern to match on
    - Wildcards (*) can be used - e.g. `on*Event`, `create*`
  - Attributes corresponding to their `@Transactional` counterparts
    - **`propagation`**: Desired TX propagation
      - default = **`REQUIRED`**
    - **`isolation`**: TX isolation level, default = **`DEFAULT`**
    - **`timeout`**: TX timeout value (sec.) default = **`-1`**
    - **`read-only`**: Is transaction read-only? default = false
    - **`rollback-for`**: Exceptions that trigger rollback; Value is comma-delimited list of classnames
    - **`no-rollback-for`**: Exceptions that DON'T trigger rollback; Value is comma-delimited list of classnames

Using Markers for Pointcuts

- Issue: TX control needed in may places
  - Pointcuts makes configuration easier
  - Still complex, and not easy to notice in the affected classes
- Alternative: Create **markers** to trigger your AOP
  - These are rubber stamps that "mark" a type for applying behavior
  - We'll illustrate marker interfaces and annotations
- Below, we defined a marker interface
  - It's an empty interface - just for use with pointcuts

```
package com.javatunes.service;

public interface ServiceTXMarker { }
```

Marker Interface for TX Control

- Below, we show `CatalogImpl` implementing the interface
- At bottom, the pointcut is defined using the marker (1)
  - Now, it's clearer in `CatalogImpl` that we're applying TX behavior
    - Even if you don't know the code conventions used, the marker interface is a good clue that something is going on
  - It's fairly easy to stamp our classes to apply the TX behavior
    - Just implement an empty interface

```
public class CatalogImpl implements Catalog, ServiceTXMarker  { … }
```

```
<aop:config>
  <aop:pointcut id="serviceOperations" expression=
             "within(com.javatunes.service.ServiceTXMarker+)" />

  <aop:advisor pointcut-ref="serviceOperations"
               advice-ref="txAdvice" />
</aop:config>
```

Marker Annotation for TX Control

- Annotations are an alternative to interfaces, e.g. **@ServiceTX** below
  - And show CatalogImpl using it
  - At bottom, the pointcut is defined using the marker (1)
- A little more complex than interfaces, but more flexible
  - Can be applied at the method level if that's useful

```
@Retention(RetentionPolicy.RUNTIME) // Much detail omitted …
@Target({ElementType.TYPE})
public @interface ServiceTX { }
```

```
@ServiceTX
public class CatalogImpl implements Catalog { /* … */ }
```

```
<aop:config>
  <aop:pointcut id="serviceOperations" expression=
             "@within(com.javatunes.service.ServiceTX)" />

  <aop:advisor pointcut-ref="serviceOperations"
               advice-ref="txAdvice" />
</aop:config>
```

Why Use Pointcut-based Configuration

- Pointcut-based Configuration of TX behavior is very powerful
  – Create pointcuts matching many methods / many classes
  – Broadly apply TX behavior in a very concise way
- Important to have **clear conventions**
  – For example, **markers** and **naming conventions**
- There are **pitfalls**
  – Changing a method name might change the TX behavior
  – e.g. from `findById` to `getById`
  – There may be no indication - other than incorrect behavior !
- You must understand AOP in general, its capabilities, and its issues to use this properly
  – But used (and documented!) properly it is very powerful

More About Pointcut Expressions

- Pointcuts are defined using **AspectJ expressions**
  - Expressions include **designators** to specify a joinpoint
  - We'll cover the major concepts here
  - We won't cover the many details - see the AspectJ docs (1)
- The **`execution`** designator has the general form shown below
  - The **return type**, **name**, and **parameter** patterns are required
  - All others are optional
  - Wildcards (e.g. *) are allowed in the patterns
  - Let's review some examples to see how it works
    - Note: The next few slides on designators are for illustration
    - Review enough to get a feel for it - not all of them

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

Sample execution Designator Patterns

- **`execution(public * *(..))`**: Matches any public method
  - `public`: the modifier pattern
  - The 1st *: the return type, and matches any type
  - The 2nd *: the method name and matches any name
  - `(..)`: Matches "any number of parameters" (zero or more)
- **`execution(* size(..))`**: Matches any invocation of `size()`
  - *: the return type (any type)
  - `size`: the method name, and matches `size` only
  - `(..)`: Matches "any number of parameters" (zero or more)
- **`execution(* com.javatunes.service.Catalog.*(..))`**: Matches invocation of a method defined in the `Catalog` interface
  - *: the return type (any type)
  - `com.javatunes.service.Catalog.*`: the method name, and matches any method in the `Catlog` interface in the given package
  - `(..)`: Matches "any number of parameters" (zero or more)

Sample execution Designator Patterns

- **`execution(* com.javatunes.service.*.*(..))`**: Matches any invocation from types in package `com.javatunes.service`
  - `com.javatunes.service.*.*`: the method name – the `*.*` means any method in any class
- **`execution(* size())`**: Matches any invocation of `size()` (from any class) that is invoked with no arguments
  - `()`: Matches "zero parameters"
- **`execution(* findByKeyword(String))`**: Matches any invocation of `findByKeyword()` that takes a `String` argument
  - `(String)`: Matches "a single String argument"
- **`execution(* findByKeyword(*, String))`**: Matches any invocation of `findByKeyword()` invoked with two arguments, with the 2nd being a `String`
  - `(*, String)`: Matches "any param, followed by a `String` param"
- **`execution(* Catalog.size(..))`**: Matches any invocation of `Catalog.size()` that is invoked with any number of arguments

Other Spring AOP Designators

- `within`: Limits matching to joinpoints within certain types
- `this`: Limits matching to joinpoints where the bean reference (Spring AOP proxy) is an instance of the given type
- `target`: Limits matching to joinpoints where the target object (application object being proxied) is an instance of the given type
- `args`: Limits matching to joinpoints where the arguments are instances of the given types
- `@target`: Limits matching to joinpoints where the class of the executing object has an annotation of the given type
- `@args`: Limits matching to joinpoints where the runtime type of the actual arguments passed have annotations of the given type(s)
- `@within`: Limits matching to joinpoints within types that have the given annotation
- `@annotation`: Limits matching to joinpoints where the subject of the joinpoint has the given annotation

Sample Designator Patterns

- `within(com.javatunes.service.Catalog+)`: Matches invocation on any object of type Catalog or its subclasses
  - + means include subclasses – needed as `Catalog` is an interface [(1)]
- `within(com.javatunes.service.*)`: Matches invocation on any type in the single package `com.javatunes.service`
- `within(com.javatunes.service..*)`: Matches invocation on any type in `com.javatunes.service` or its subpackages
  - `..` mean include subpackages
- `target(com.javatunes.service.Catalog)`: Match invocation where the target object (not the proxy) is a `Catalog` instance
- `@annotation(com.javatunes.aspects.Loggable)`: Match invocation where target is annotated with `@Loggable`
- `args(String)`: Match invocation on any method with a String arg
- `args(*)`: Match on method with a single arg of any type

[Demo] Lab 6.2: Pointcut-Based TX

In this demo lab, we illustrate using markers and pointcuts to control transactions in our service layer

Review Questions

- What are Spring transaction managers?
- What is declarative transaction management, and how does it work?
- How do you use declarative transaction management in Spring?

Lesson Summary

- Spring transaction managers **abstract your program** from the underlying transaction resource
  - Transaction managers for most major environments
  - JDBC connection, JTA transaction manager, JPA, etc.
- Spring supports **declarative specification** of TX behavior
  - Simplifies coding greatly
  - TX propagated by container from one bean to another based on their TX attributes
  - Attributes include: propagation, isolation, timeout, read-only, rollback-for, no-rollback-for
- Declare TX attributes with **annotations** or **Pointcut-based Configuration**
  - Using **Pointcut-based Configuration plus markers** is very powerful and maintainable

Session 7: Web Applications and Spring MVC

Integration with Java EE

Spring MVC Basics

View Resolvers

Controller Details

Forms and Model Objects

Lesson Objectives

- Integrate the Spring container with regular Java Web Applications (first section)
- Understand what Spring MVC is, its goals, and its architecture
  - Learn how to configure the Spring MVC `DispatcherServlet`
  - Use Command Controllers to handle requests
  - Use View Resolvers to map logical names to actual resources
    - e.g. to JSP views
  - Use Spring MVC's support for forms
    - Including model (command) classes and form controllers
  - Understand how Handler Mappings map URLs to controllers

Integration with Java EE

Integration with Java EE

Spring MVC Basics

View Resolvers

Controller Details

Forms and Model Objects

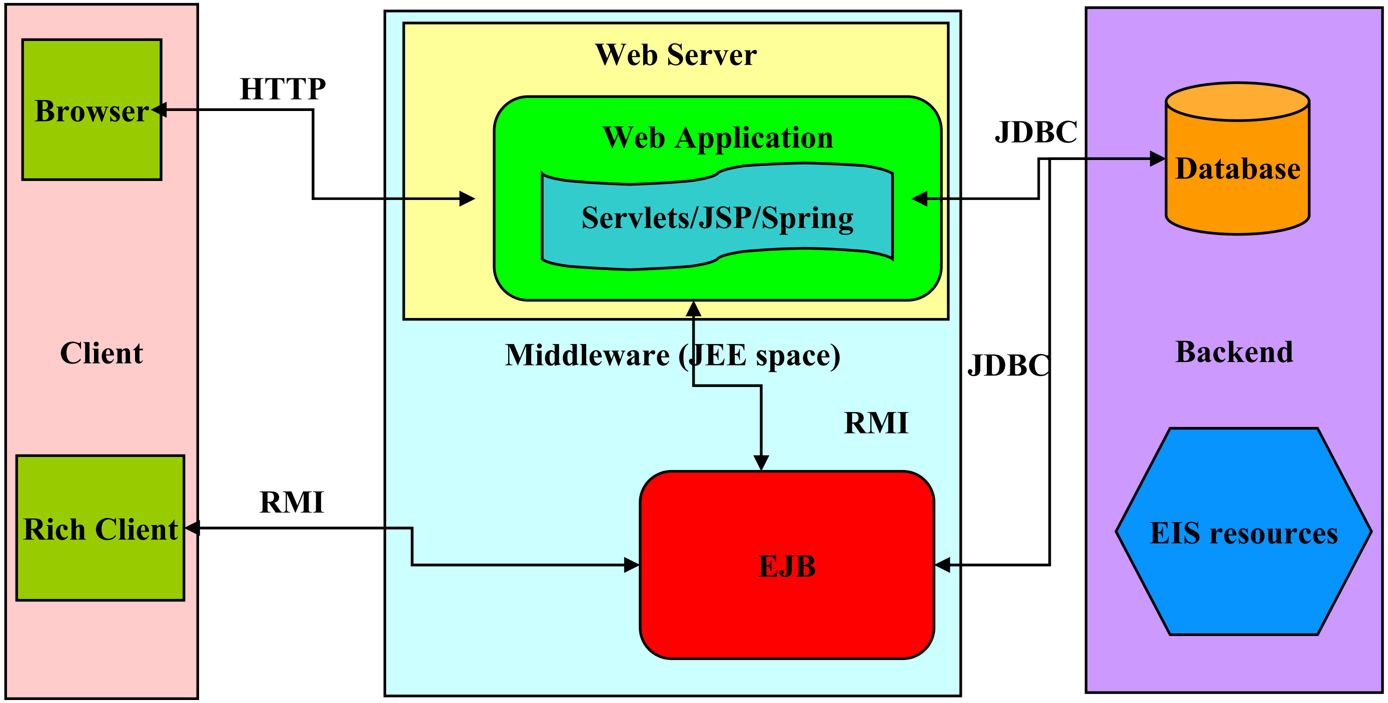
Spring and Java Enterprise Edition (JEE)

- **JEE**: Standard Java architecture for building scalable, distributed, reliable, Enterprise apps
  - Enterprise Edition basically means "Server Side" Java

- Spring can complement or replace many parts of JEE
  - Often useful to integrate existing web apps with Spring
  - Fairly easy to use **Spring beans and DI** with JEE Web apps
    - We'll illustrate
  - Used for Spring MVC integration also
    - Via the Spring MVC dispatcher servlet covered later

Overview of JEE Web Applications

- Standard structure defined in JEE
  - Collection of **Servlets**, **JavaServer Pages** (JSP), & other files
    - Or JSF controllers and views
  - Most frequently collected inside of a **WAR** (Web Archive) file
    - The standard JEE web app packaging



Web Application Structure

- Java web apps are organized in the standard structure below

- **<Web application base directory>**
  - [static content files: HTML, forms, images, etc.]
  - [dynamic content: e.g. JSP]
  - [other content directories]
  - **WEB-INF**
    - **web.xml**
    - **classes**
      - [.class files: servlets and others]
    - **lib**
      - [JAR files]

Web application configuration file

Web Application Components

- **Servlets**: Java components that run on the server
  - Servlets run **in response to client requests**
  - Defined by extending class `javax.servlet.HttpServlet` and overriding `doGet()/doPost()`
- **JSP**: JavaServer Page - contains dynamic and static content
  - **Template Data** consists of regular text, HTML, XML ...
  - **Dynamic Data** is generated anew for each request - e.g from custom tags in the JSP
- **web.xml**: Deployment descriptor for web app
  - Contains configuration information for the web app
  - For example, the servlets and their mappings
  - Optional in current releases of JEE

ApplicationContext and Java Web Apps

- Spring's **`WebApplicationContext`** integrates with the JEE Web container
  - It's a context tailored for Web app use

- Load context easily with Spring's **`ContextLoaderListener`**
  - Configure in *web.xml* - loads context on web app deployment
  - Use `<context-param>` to configure properties, including
    - Configuration file locations
    - Use of XML-based or Java-based config

- Java Web apps easily access the context via **`WebApplicationContextUtils`**
  - Via static **`getWebApplicationContext()`**

Configuring ContextLoaderListener - XML

- We configure `ContextLoaderListener` in *web.xml* below
  - Defaults to XML-based configuration
  - The `contextConfiguration` param provides config file locations [1]
  - *applicationContext.xml* and *repository.xml* in this example

```
<web-app ... >

<context-param>  <!--  config files for ContextLoaderListener -->
  <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/applicationContext.xml <!-- Default location -->
      /WEB-INF/repository.xml
    </param-value>
</context-param>

<listener>  <!-- Load root application context at startup -->
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

</web-app>
```

ContextLoaderListener - @Configuration

- Below, we configure the listener to use Java-based config
  - `contextClass` is specified as **`AnnotationConfigWebApplicationContext`**
  - `contexConfigLocation` now specifies the **config class**(es) (1)

```
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
   org.springframework.web.context.support.AnnotationConfigWebApplicationContext
  </param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>com.javatunes.config.SpringConfig</param-value>
</context-param>

<listener> <listener-class> <!-- Load root application context at startup -->
      org.springframework.web.context.ContextLoaderListener
</listener-class> </listener>
```

Using the Application Context

- Below, `WebApplicationContextUtils` retrieves the context
    - By doing a simple lookup on the Servlet context via a known name
  - The context was loaded by the `ContextLoaderListener`
  - Bean lookup is same as we've seen before
    - Example assumes same registered beans as earlier examples
- Integrating Spring with Web apps is easy
  - For other frameworks (JSF, Spring MVC) integration is even easier [1]

```
import org.springframework.web.context.support.WebApplicationContextUtils;
import org.springframework.web.context.WebApplicationContext;

public class SearchServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {

  WebApplicationContext ctx =
      WebApplicationContextUtils.getRequiredWebApplicationContext(
                                                       getServletContext());
  Catalog cat = ctx.getBean(Catalog.class);
}
```

LEARNINGPATTERNS

Lab 7.1: Spring and the Web

In this lab, we will integrate the Spring container with a regular Java Web application

Spring MVC Basics

Integration with Java EE

Spring MVC Basics

View Resolvers

Controller Details

Forms and Model Objects

What is Spring MVC?

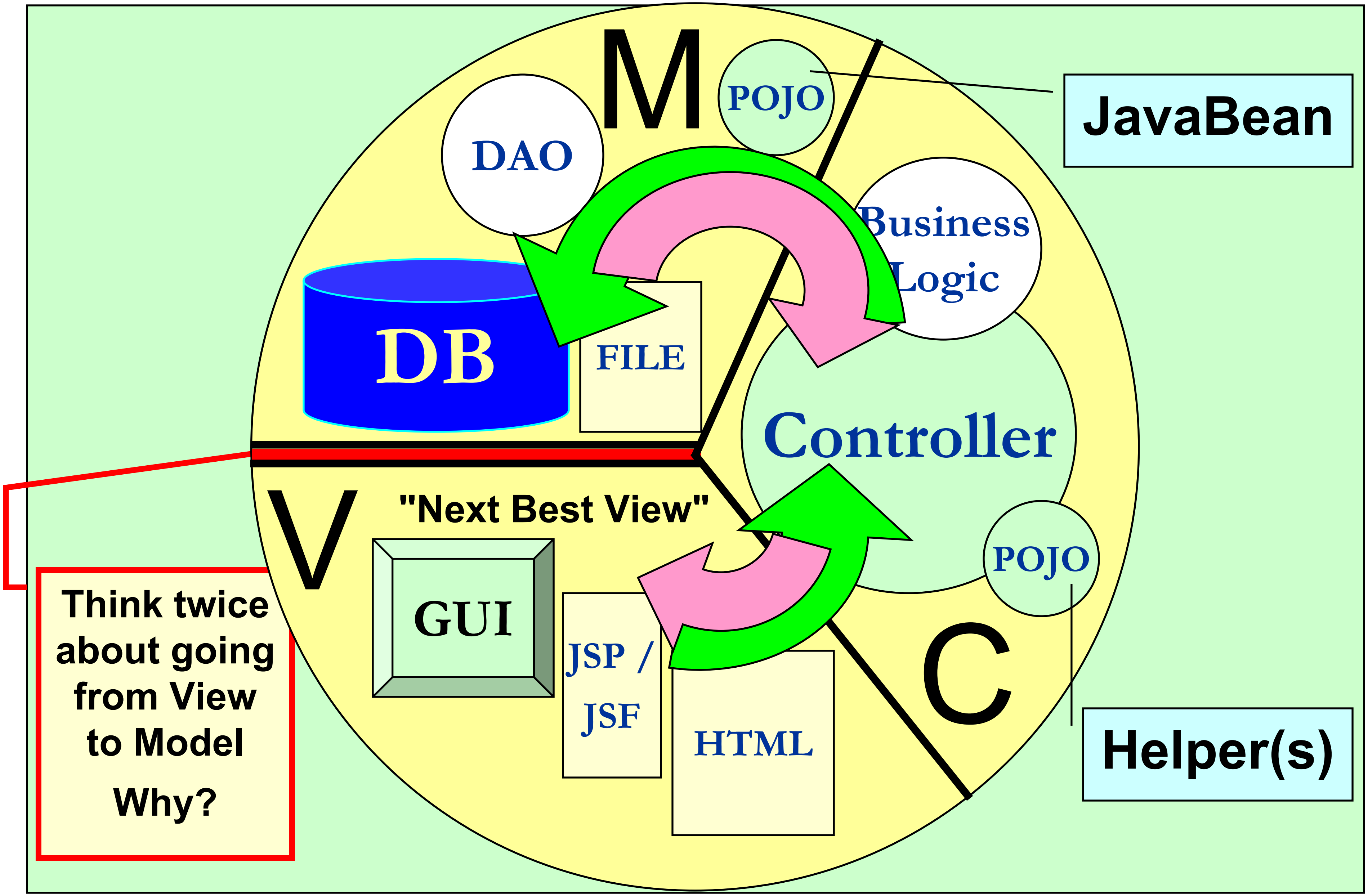
- Java Web app framework based on **M**odel **V**iew **C**ontroller (MVC)
  - Roughly equivalent to JSF (JavaServer Faces) in capability [1]
  - Viable alternative for Spring users

- Overall Spring MVC goals
  - **Simplify** web app creation
  - Provide a **strong MVC framework** with clearly defined roles
    - Controller, command object, view resolver, etc.
    - Encourages good architecture
  - **Integrate** easily with the rest of Spring
  - Build on **standard Java EE** (servlets/JSP, custom tags)
  - Flexibly integrate with additional template technologies (e.g. Velocity)

General MVC Architecture

- **M**odel **V**iew **C**ontroller
  - Standard pattern for applications with user interfaces
- **Model**: Represents the business domain
  - Independent of user interface or application flow
  - Data and business objects - may be Spring-based or not
- **View**: Presents data to a human user
  - Generally producing HTML as an end product
  - Often a **JSP** or **JSF** page, plus other resources (images, etc.)
- **Controller**: Intermediary between Model and View
  - Controls user flow through the app, and invokes business logic
  - Can be implemented as servlets in vanilla JEE
  - Many frameworks provide specialized controllers

MVC Pattern flow

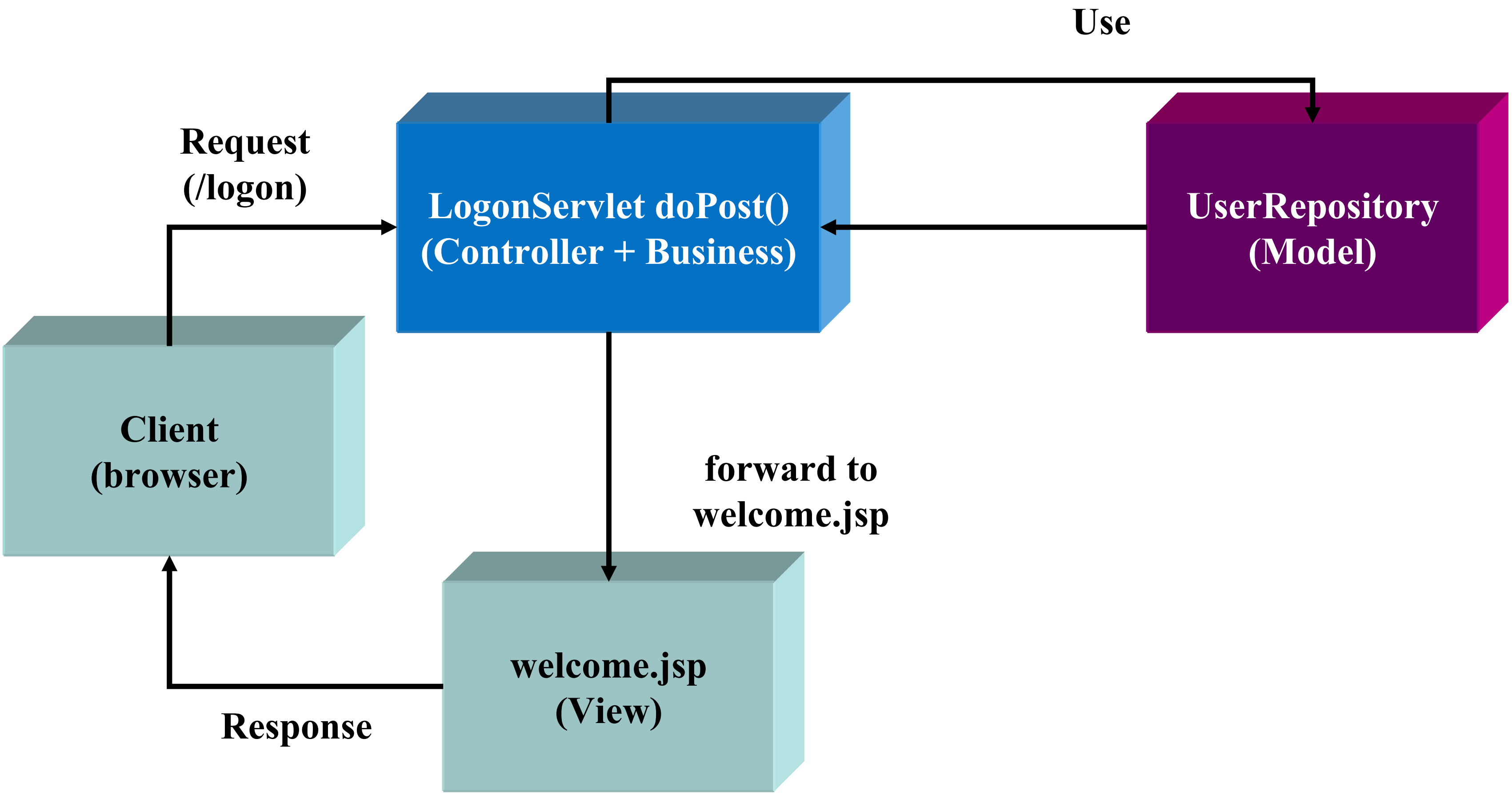


Spring MVC Architecture

- Based on **front controller** architecture
  - Front controller receives **all requests**
  - **Dispatches requests** to other components to handle

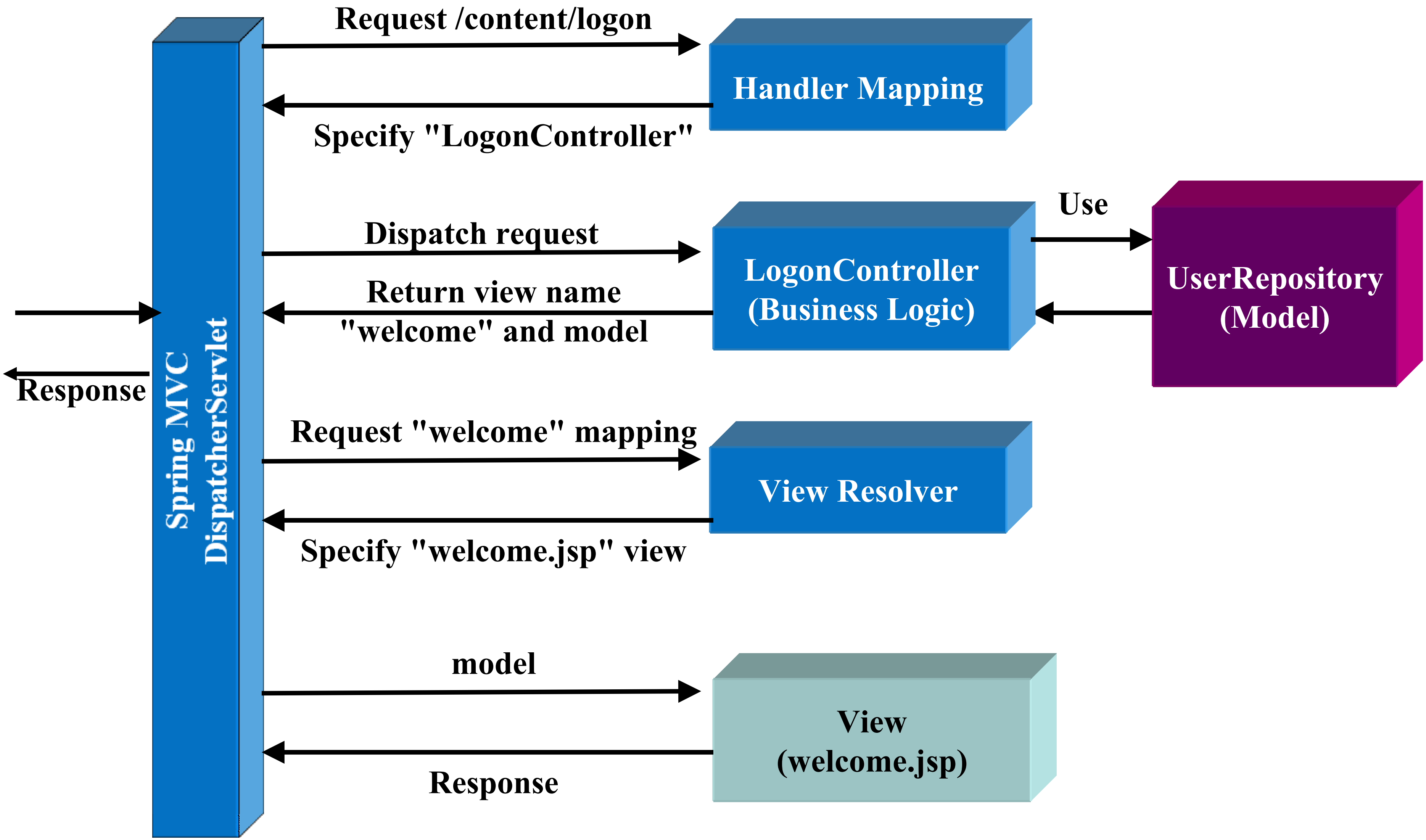
- Spring MVC components include:
  - **`DispatcherServlet`**: The front controller servlet
  - **`Controllers`**: Handle specific requests, generate response data, select a view
  - **Handler Mappings**: Map request URLs to resources/controllers
  - **Views**: Generate HTML Output (via JSP or other templates)
  - **View Resolvers**: Process response from controllers and direct the application to the appropriate view (e.g. JSP page)
  - **Model**: Spring managed beans

Plain Servlets/JSP Request Flow



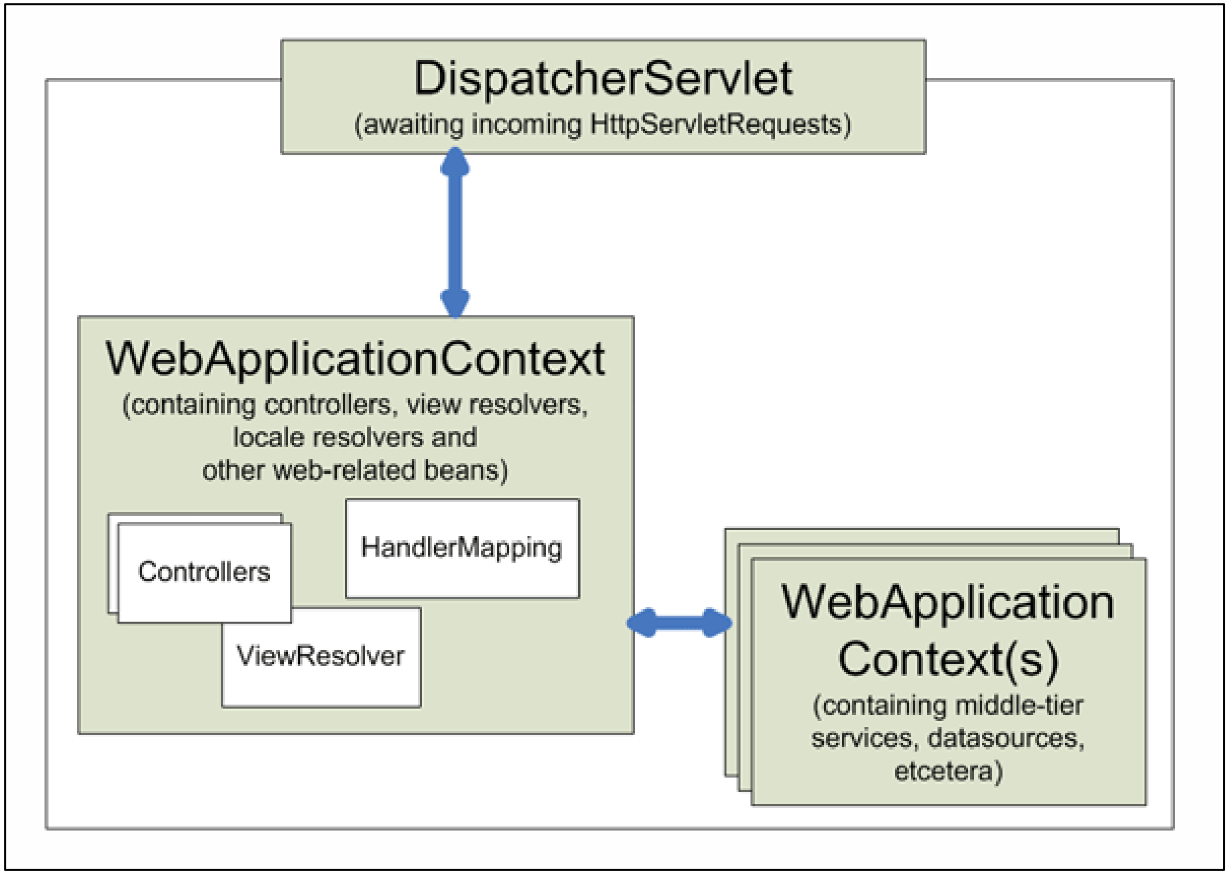
Spring MVC Request Flow

- The `DispatcherServlet` is the **front controller**



DispatcherServlet Initialization

- The `DispatcherServlet` has its own `WebApplicationContext`
  - Initialized by the framework when the servlet is initialized
  - We'll see how to initialize this context next



DispatcherServlet - Java Config

- Spring supports Servlet 3 capabilities for an an all-Java config
  - Example uses `SpringConfig.class` for standard Spring config
    - Root context with shared resources - e.g. service objects
  - Uses `WebConfig.class` for Spring MVC-specific config

```
// See notes for import (1)
public class JavaTunesWebAppInitializer extends
       AbstractAnnotationConfigDispatcherServletInitializer {

  @Override  // URLs that are routed to dispatcher
  protected String[] getServletMappings() {
    return new String[] { "/content/*" };
  }
  @Override // Config class for root context
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] { SpringConfig.class };
  }
  @Override // Config class for Web app context
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] { WebConfig.class };
  }
}
```

Spring MVC Configuration - Java Config

- Below, we configure Spring MVC using `@Configuration` style
  - **`@EnableWebMvc`**: Initializes Spring MVC (registers related beans [1])
  - **`@ComponentScan("com.javatunes.web")`**: Standard annotation
    - Not specific to Spring MVC
    - We're using it here to configure auto-scanning of Spring MVC controllers
- We don't show `SpringConfig.class`
  - Similar to examples seen earlier

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

// Configure Spring MVC support
@Configuration
@EnableWebMvc
@ComponentScan("com.javatunes.web")
public class WebConfig {}
```

DispatcherServlet - web.xml Config

- Configured as standard servlet to map requests to the dispatcher (1)
  – Typically, use a pattern match such as ***/content/****
  – Requests having the prefix (e.g. ***/content/logon***) go to Spring MVC
  – We also configure the name of the web app context config file (2)

```
<!-- ContextLoaderListener not shown - as before for root context -->

<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/configuration/webmvc.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <url-pattern>/content/*</url-pattern>
</servlet-mapping>
```

Spring MVC Configuration - XML Config

- Below, we configure Spring MVC using XML style

  <mvc:annotation-driven/>

  - Initializes Spring MVC by registering related beans (1)

  <context:component-scan base-package="com.javatunes.web"/>

  - Discovers beans via standard auto-scan we've seen before

```
<beans … > <!-- Namespaces not shown - see notes for them -->

  <!-- Component scanning for controllers -->
  <context:component-scan base-package="com.javatunes.web"/>

  <!-- Initializes Spring MVC support -->
  <mvc:annotation-driven/>

</beans>
```

Mini-Lab: DispatcherServlet Reference

Mini-Lab

- We provide the Spring Reference under
  - Under *StudentWork/Spring/**Resources/SpringDocs/reference***
  - In a browser, open *index.html* in the folder above (if not already open)
- Click on the link for the "**Web Servlet**" reference documentation
  - In the left hand column, click the link for the **Spring Web MVC**
    - Currently **Section 1**
  - Find Section **1.2.1** Context Hierarchy (under Dispatcher Servlet)
- Spend 5 minutes reviewing this section
  - It explains both the context hierarchy and the configuration details

Controllers

- **Controllers**: **Handle requests** in Spring MVC
  - Dispatcher servlet forwards requests to controllers
  - Interprets user input, generates a model, and directs response to a view
  - Shields you from low level APIs (e.g. servlet API)

- They're **annotation-based** POJOs, and the API includes:
  - **`@Controller`**: Marks class as a web controller
    - Stereotype of `@Component` - usually auto-detected via scanning
    - In package **`org.springframework.stereotype`**
  - **`@RequestMapping`**: Maps requests to a controller class/method
    - In package **`org.springframework.web.bind.annotation`**
  - **`RequestMethod`**: enum with HTTP request methods (e.g. GET)
  - **`@GetMapping, @PostMapping, @PutMapping`**, etc.
    - Shortcuts for specific HTTP methods, e.g.

    `@GetMapping == @RequestMapping(method = RequestMethod.GET)`

- **Component scanning** must be configured to use these

A Very Simple Controller

- `LogonController` receives requests like **`/content/logon`**
  - **`@Controller`** indicates it's a controller
  - **`@RequestMapping("/logon")`** specifies the base URL handled
  - **`@RequestMapping(method = RequestMethod.GET)`**: Specifies that `get()` handles HTTP GET requests
  - **`@ResponseBody`**: Indicates return value is actual view content
    - i.e. HTML - we'll do this better shortly (1)

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

@Controller
@RequestMapping("/logon")
public class LogonController {

  @RequestMapping(method = RequestMethod.GET)
  @ResponseBody // Specify that we return the actual response (1)
  public String get() { return "<h1>Hello Spring MVC</h1>"; }
}
```

Web App Flow for Logon

- A `/content/logon` request results in the following flow
  - **`DispatcherServlet` receives** the request
    - Based on the **`/content/*`** mapping for the dispatcher servlet
  - It forwards the request to **`LogonController.get()`**
    - Based on the `@Controller` and `@RequestMapping` in the controller
    - `get()` returns literal view content to the controller (because of `@ResponseBody`)
  - `DispatcherServlet` generates the resulting view
    - Using the content from the controller method
    - Generally, we'll use logical view names and JSP pages instead
    - We'll see this being done shortly
- That's it for this simple controller

@RequestMapping Defines Controllers

- **@RequestMapping** on a controller **class** defines a **root path**
  - All requests handled by the controller are below this path
  - Below, we set the root path to **/catalog**
  - Full path to this controller is **/content/catalog**
- The path can include ant-style path patterns
  - e.g. **/catalog/*/onSale**
- The path can also include **${…}** placeholders
  - For local or system properties and environment variables

```
// Imports omitted
@Controller
@RequestMapping("/catalog")
public class CatalogController {
  /* … */
}
```

Handler Methods

- **`@RequestMapping`** on a controller **method** designates a **handler method**
- Handler methods can specify:
  - The **HTTP method** (default - all HTTP methods)
  - An additional **sub-path** to qualify the URI
  - Below, `getItems()` handles requests to ***/content/catalog/items***
    - Handles **GET** requests only
- Handler methods can also take parameters (covered soon)

```
@Controller
@RequestMapping("/catalog")
public class CatalogController {

  @RequestMapping(value="/items", method = RequestMethod.GET)
  public String getItems() { /* … */ }
}
```

Mini-Lab: Controller Reference

Mini-Lab

- Continuing in the Spring Reference docs, under the Web Servlet section
  - Find Section 1.4.1 (**Annotated Controllers**)
- Spend a couple of minutes reviewing this section

Lab 7.2: Spring MVC Basics

In this lab, we'll set up Spring MVC, and use some basic capabilities to create a simple controller

View Resolvers

Integration with Java EE

Spring MVC Basics

View Resolvers

Controller Details

Forms and Model Objects

View Resolvers

- Controllers generally return **logical view names** (e.g. `welcome`)
  - Which are then mapped to views (e.g. */views/welcome.jsp*)

- **View Resolvers** handle this in Spring MVC
  - They're classes that resolve logical names to views
  - Used by the dispatcher to determine the next view
    - Which may be a JSP page, a FreeMarker template, etc.

- **`InternalResourceViewResolver`** is a common implementation
  - In package: `org.springframework.web.servlet.view`
  - Adds a **prefix** and **suffix** to a logical name to generate view URL
    - These are set via configuration
  - Forwards request to the URL via `RequestDispatcher`
  - Suitable for view resources within the Web app (1)

View Resolvers - Java Config

- Configured using **`interface WebMvcConfigurer`**
  - In `org.springframework.web.servlet.config.annotation`
  - Defines callback methods for customizing Spring MVC
    - Configures view resolvers, formatters, interceptors, and more

- In Spring 5, the interface has Java 8 defaults for all methods
  - Just implement the interface in your `@Configuration` class
    - Then override the methods you need
  - In earlier Spring versions, extend `WebMvcConfigurerAdapter` (1)

- To configure a view resolver, declare a bean of type `ViewResolver`
  - We illustrate on next slide using an `InternalResourceViewResolver`

Example: View Resolvers - Java Config

- Below, we configure an `InternalResourceViewResolver` bean
  - Detected and used automatically (see notes for XML config)
  - We set the prefix to **`/views`**, and the suffix to **`.jsp`** (JSP pages)
  - It's also common to place view pages under WEB-INF (1)

```
@Configuration
@EnableWebMvc
@ComponentScan("com.javatunes.web")
public class WebConfig implements WebMvcConfigurer { // import omitted

  @Bean   // Configure a view resolver bean
  public ViewResolver viewResolver() {
    InternalResourceViewResolver resolver =
      new InternalResourceViewResolver();
    resolver.setPrefix("/views/"); // Set the prefix
    resolver.setSuffix(".jsp");    // Set the suffix
    // Expose all beans in the context to your view pages (2)
    resolver.setExposeContextBeansAsAttributes(true);
    return resolver; // Return the resolver
  }
}
```

Java Config Simplified Configuration

- **`WebMvcConfigurer`** supports simplified view resolver configuration
  - Via a callback and **`ViewResolverRegistry`** - illustrated at bottom
  - Simpler, and contains methods for many common view resolvers - e.g. JSP, Freemarker, BeanName, etc.
  - Doesn't always support all needed configuration
    - In that case, create the `ViewResolver` bean directly as seen previously

```
@Configuration
@EnableWebMvc
@ComponentScan("com.javatunes.web")
public class WebConfig implements WebMvcConfigurer {

  @Override
  public void configureViewResolvers(ViewResolverRegistry registry) {
    registry.jsp("/WEB-INF/views/", ".jsp");
  }
}
```

Controller Using Logical Names

- Controllers now use **logical view names** (not file names)
  - Illustrated at bottom
  - **`"logon"`** is mapped by our view resolver to **`/views/logon.jsp`**
  - Use a `redirect:` prefix if you want a redirect (1)
    - Instead of a forward
- Note that this `get()` handler generates the initial logon view page
  - It's common to do this through a handler (and not go directly to a JSP)
  - Provides for initialization of model objects (covered soon)
  - Also hides file names from the client

```
@Controller
@RequestMapping("/logon")
public class LogonController {

  @RequestMapping(method = RequestMethod.GET)
  public String get() { return "logon"; }

}
```

Web App Flow Revisited

- A request to **`/content/logon`** results in the following flow
  - i.e. a request to http://myhost.com/myapp/**content/logon**
  - **`DispatcherServlet` receives** the request
    - Based on the **`/content/*`** mapping for the dispatcher servlet
  - It forwards the request to **`LogonController.get()`**
    - Based on the `@Controller` and `@RequestMapping` in the controller
    - `get()` returns a value of "`logon`"
    - This is mapped to `/views/logon.jsp` by the view resolver
  - `DispatcherServlet` generates the resulting view
    - Using the page `/views/logon.jsp`

Other View Resolvers

- **`BeanNameViewResolver`**: Maps view name to a Spring bean
  - Renders views with a Java class (not a JSP)
  - Example maps the view name "`stats`" to the SpreadsheetView bean (1)
- **`XmlViewResolver`**: Uses external XML file to map views to beans
  - Externalizes bean mappings for views to an XML file
  - Default XML file is */WEB-INF/views.xml*
- **`ResourceBundleViewResolver`**: Uses external properties files to map views to beans
  - Allows for localization / internationalization
  - We'll illustrate how this one works

```
<!-- Configure a bean name view resolver -->
<bean id="viewResolver"
  class="org.springframework.web.servlet.view.BeanNameViewResolver"/>

<!-- WelcomeView class used to generate view for view name "welcome" -->
<bean id="stats" class="com.javatunes.web.SpreadsheetView"/>
```

Example: ResourceBundleViewResolver

- Uses a properties file (on the classpath) for view resolution
  - Default filename: **`views.properties`**
  - Set via `basename` property (e.g. to **`javatunes-views`**)
  - Works with standard I18N support in Java properties files
    - i.e. will use *javatunes-views_**de**.properties* if locale is **de**
- Sample configuration below (`@Configuration` and XML)

```
@Bean  // Appears in your WebMvcConfigurer class
public ViewResolver viewResolver() {
 ResourceBundleViewResolver resolver = new ResourceBundleViewResolver();
 resolver.setBasename("javatunes-views");
 return resolver;
}
```

```
<bean id="viewResolver"
    class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="baseName"><value>javatunes-views</value></property>
</bean>
```

Properties File - ResourceBundleViewResolver

- Below, we configure view mappings in the properties file
  - Two view types are illustrated
  - **`JstlView`**: For JSP pages using JSTL
  - **`RedirectView`**: Redirects to a URL (Sends redirect to browser)
- The view mappings are
  - **`login`** and **`welcome`**: JSP views using *logon.jsp* / *welcome.jsp*
  - **`logOffRedirect`**: Configured as a redirect going to the logon view

```
# /WEB-INF/classes/javatunes-views_en.properties
# Assumes locale of en

login.(class)=org.springframework.web.servlet.view.JstlView
logon.url=/WEB-INF/jsp/logon.jsp

welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

logOffRedirect.(class)=org.springframework.web.servlet.view.RedirectView
logOffRedirect.url=logon
```

Mapping DispatcherServlet to /

- Gives simpler URLs - e.g. **/logon** instead of /content/logon
  - Configured as shown below (Replaces default servlet handling)
- You must also enable content handling for the default servlet
  - For requests the dispatcher servlet doesn't handle - e.g. image, JSP, etc.
  - Configure as shown at bottom (See note (3) for XML equivalents)

```
// In JavaTunesWebAppInitializer - other detail not shown (1)
  @Override  // URLs that are routed to dispatcher - use root
  protected String[] getServletMappings() {
    return new String[] { "/" };
  }
```

```
// In WebConfig - other detail not shown (2)
  @Override
  public void configureDefaultServletHandling(
                          DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
  }
```

Controller Handling /

- Below, we show our controller modified to handle /
  - Requests to the root (/) are now routed here
  - This controller handles 2 URLs, as we've kept the /logon handling
- Now it handles both of the following
  - http://myhost.com/myapp/logon
  - http://myhost.com/myapp/

```
@Controller
@RequestMapping({"/", "/logon"})
public class LogonController {

  @RequestMapping(method = RequestMethod.GET)
  public String get() { return "logon"; }

}
```

Lab 7.3: View Resolvers

In this lab, we set up and use a view resolver

Controller Details

Integration with Java EE

Spring MVC Basics

View Resolvers

Controller Details

Forms and Model Objects

Request Parameters and Model Data

- Controllers often:
  - – Extract input data from the request
  - – Return model data to the view for rendering

- Spring supports this via:
  - – **Binding request parameters** to handler method args via **@RequestParam**
    - In `org.springframework.web.bind.annotation`
  - – **Populating model objects** in the handler methods
    - We'll look at two ways to do this
    - Views use data from model objects
  - – There are many more techniques - we'll look at a few of them

Sample Input Pages

- Assume a logon form and resulting welcome page as below
  - Plain JSP pages (no Spring functionality)
- The logon form:
  - Submits to /content/**logon**
  - Includes **name** and **password** parameters
- The welcome page uses a `Principal` object
  - The handler will add this to the model

```
<!-- Logon submission page -->
<form method='post'
       action='${pageContext.request.contextPath}/content/logon'>
  <input size='20' type='text' name='name'/>
  <input size='20' type='password' name='password'/>
  <input type='submit' name='Submit' value='Logon'/>
</form>
```

```
<!-- Welcome page welcome.jsp -->
Welcome ${principal.name}
```

@RequestParam – Parameter Binding

- Below, we show a **POST** handler for LogonController
  - It binds the name/password request parameters to method args
- **@RequestParam** binds request parameters to handler method args
  - Can be left out if arg name matches request parameter name
- **`@RequestParam`** has two other optional elements (see notes)
  - **`required`**: Whether parameter is required
  - **`defaultValue`**: Default value if no value on request

Example: **`@RequestParam(value="password", required="true")`**

```
// Part of LogonController - other detail omitted
@RequestMapping(method = RequestMethod.POST)
public String processLogon(
  @RequestParam("name") String name,
  @RequestParam("password") String password) {
  // Logon processing detail not relevant and omitted …
  return "welcome";
}
```

Returning Model Data

- Handler methods can return model data to the view
  - Generally using model classes
  - Two techniques for this described below

- **Include a `Model` argument** in the handler method
  - In `org.springframework.ui`
  - Initialized by the container when the handler called
  - Your handler code can populate it with model data

- **Return a `ModelAndView`** from the handler (instead of a string)
  - `org.springframework.web.servlet.`**`ModelAndView`**
  - Contains both the logical view name, and the model data
  - You create the instance in your handler and return it

Class ModelAndView

- `ModelAndView` holds both view information and model data
  - Usable as a handler return value
  - Has constructors/methods to get/set view, add attributes to the model
  - Some common constructors / methods include:

```
ModelAndView(String viewName, String modelName,
  Object modelObject)
```

  - Creates instance with given view name, model name, and model data

```
ModelAndView(String viewName, Map<String,?> model)
```

  - Holds multiple model objects in the map

```
ModelAndView addObject(String attributeName,
  Object attributeValue)
```

  - Add an attribute to the model with the given name

  - See notes for more methods

Returning ModelAndView Example

- Below, the handler creates a `ModelAndView` and returns it
  - We initialize it with the view name and model data
- `DispatcherServlet` does the following
  - Extracts the data from the model map
  - Makes model attributes available to the view
    - For JSP, it puts them on the request
    - They can be accessed via JSP EL variables
  - Forwards to a view page based on the view value

```
@RequestMapping(method = RequestMethod.POST)
public ModelAndView processLogon(
  @RequestParam("name") String name,
  @RequestParam("password") String password) {
  // Assume checkLogon defined elsewhere - detail not shown
  Principal p = checkLogon(name,password);
  return new ModelAndView("welcome", "principal", p);
}
```

Model Argument Example

- Below, our handler method includes a Model argument
  - This is initialized by the container before it's called
  - Our handler adds the model data into this object
  - It then returns a view string, as previously

```
@RequestMapping(method = RequestMethod.POST)
public String processLogon(
 @RequestParam("name") String name,
 @RequestParam("password") String password,
 Model model) {
   // Assume checkLogon defined elsewhere - detail not shown
   Principal p = checkLogon(name,password);
   model.addAttribute("principal", p);
   return "welcome";
}
```

Other Handler Method Capabilities

- Handler methods support many arguments that are automatically initialized, including:
  - **Request** or **response** objects from Servlet API
  - **Session** object: The `HttpSession`
  - `java.util.`**`Locale`**: Current request Locale
  - **`InputStream`**/**`Reader`**: Raw stream/reader exposed by servlets
  - **`OutputStream`**/**`Writer`**: Raw stream/writer exposed by servlets
  - **`@RequestHeader`** annotated parameter: Specific header
  - **`@CookieValue`** annotated parameter: Specific cookie

- Return value types include:
  - **`ModelAndView`**, as seen earlier
  - **`Model`** object, with view name implicitly generated from requests
  - **`Map`** object, with view name implicitly generated from request
  - **`View`** object, with model implicitly determined

Example – Other Handler Capabilities

- In the example below, we show an example of using some of the additional capabilities of handler methods

```
// Access HttpServletRequest and HttpServletResponse Objects
@RequestMapping("/getServletObjects")
public void displayServletInfo(
  HttpServletRequest request, HttpServletResponse response)
  { System.out.println(request.getParameter("password"); }


// Access session object and cookie containing session id
@RequestMapping("/displaySessionInfo")
public void displaySessionInfo(
  HttpSession theSession, @CookieValue("JSESSIONID") String sessID)
  { /* … */ }


// Access request header values
@RequestMapping("/displayHeaderInfo")
public void displayHeaderInfo(
   @RequestHeader("Accept-Encoding") String encoding,
   @RequestHeader("Keep-Alive") long keepAlive) { /* … */ }
```

Lab 7.4: Client Input / Model Data

We use client input, and return model data to the view

Review Questions

- How do you integrate the Spring container with normal web applications?
- What is MVC?
- What are the architecture and main components of Spring MVC?
- How do you configure Spring's `DispatcherServlet`?
  – What does its URL mapping look like?
- What does a Spring MVC controller do?
  – How is it invoked?

Lesson Summary

- Spring integrates with Java Web apps via Springs **`ContextLoaderListener`**
  - Loading Spring's root application context and making it available to Web apps
- **MVC**: A common pattern for organizing apps with user interfaces into three areas of functionality
  - **Model**: Representing the business domain
  - **View**: Presenting the data to users
  - **Controller**: An intermediary between the model and the view
- Spring implements MVC as follows:
  - Spring beans used as model
  - Normal view pages (e.g. JSP) as view
  - Front controller dispatching to controller classes/methods

Lesson Summary

- Spring's `DispatcherServlet` is a normal servlet configured in *web.xml* or in a Servlet 3 initializer
  - Its URL mapping uses patterns, e.g. `/content/*` or `*.do`
  - This mapping is applicable to a whole category of requests
  - It acts as a front controller to the web app
- **Controller classes/methods** handle incoming requests
  - They access request data, execute business logic, set up the model with data, and indicate the next view to render
- **Annotating** controllers makes configuration very simple

Session 8: More Spring MVC Capabilities

Forms and Model Objects

Working with Sessions

Validation

Lesson Objectives

- Use Spring MVC's support for forms including model classes and Spring's tag libraries

- Learn how to use HTTP sessions with Spring MVC

- Be familiar with JSR-303 validation

Forms and Model Objects

Forms and Model Objects

Working with Sessions

Validation

Model Classes

- Spring MVC can use POJOs for the model
  - To hold both client input and response data
  - The `Logon` class below holds a logon name, password, and a `Principal` object (the result of a successful logon)
  - It's a POJO (following JavaBean property naming conventions)

```
package com.javatunes.web;
import java.security.Principal;

public class Logon {
  private String name;
  private String password;
  private Principal principal;

  public String getName() {return name;}
  public void setName(String nameIn) {this.name = nameIn;}
  // Other get/set methods not shown
}
```

@ModelAttribute

- **`@ModelAttribute`** on a handler argument indicates it is retrieved from the model
  - If it's not on the model, a new instance is created, and added to it
  - The model is then available to views
- Below, we bind an instance of Logon into the model
  - We initialize the name (part of the model data) in `get()`
  - It's now available to the view (*logon.jsp*)
  - We'll use Spring's custom `form:` tags to access it

```
@RequestMapping(method = RequestMethod.GET) // In LogonController
public String get(@ModelAttribute("logon") Logon l) {
  l.setName("Jane Doe");  // Can initialize the model if you want
  return "logon";  // Return the view name
}
```

Spring MVC Form Tags <form:form>

- **`<form:form>`** renders an HTML `<form>` and exposes a model object as a **binding path** to inner tags for binding
  - Inner tags access properties relative to this binding path
    - **`modelAttribute='logon'`** exposes the logon object within the form (1)
    - It puts the object in the `PageContext` for access by inner tags
  - This initial form view **must** be generated via a controller
    - `LogonController.get()` directs a GET of */views/logon* to this page
    - The form's action is generated from the URL associated with that controller

```
<!-- Logon submission page /views/logon.jsp -->
<%@ taglib prefix="form"
         uri="http://www.springframework.org/tags/form" %>
<form:form modelAttribute='logon'>
  <!-- Other input fields shown soon -->
  <input type='submit' name='Submit' value='Logon'/>
</form>
```

Spring MVC Form Tags <form:input>

- **<form:input>** renders an HTML `<input>`
  - And default of `type="text"`
  - Supports other HTML 5 types, e.g. `type="email"`

  - **`path='name'`** binds the input element to the **`logon.name`** property
    - The path is relative to the binding path set up in the enclosing `<form:form>`

  - **`path='password'`** binds the input to the **`logon.password`** property

```
<!-- Logon submission page /views/logon.jsp -->
<%@ taglib prefix="form"
         uri="http://www.springframework.org/tags/form" %>
<form:form modelAttribute='logon'>
  <form:input size='20' path='name'/>
  <form:input size='20' path='password'/>
  <input type='submit' name='Submit' value='Logon'/>
</form>
```

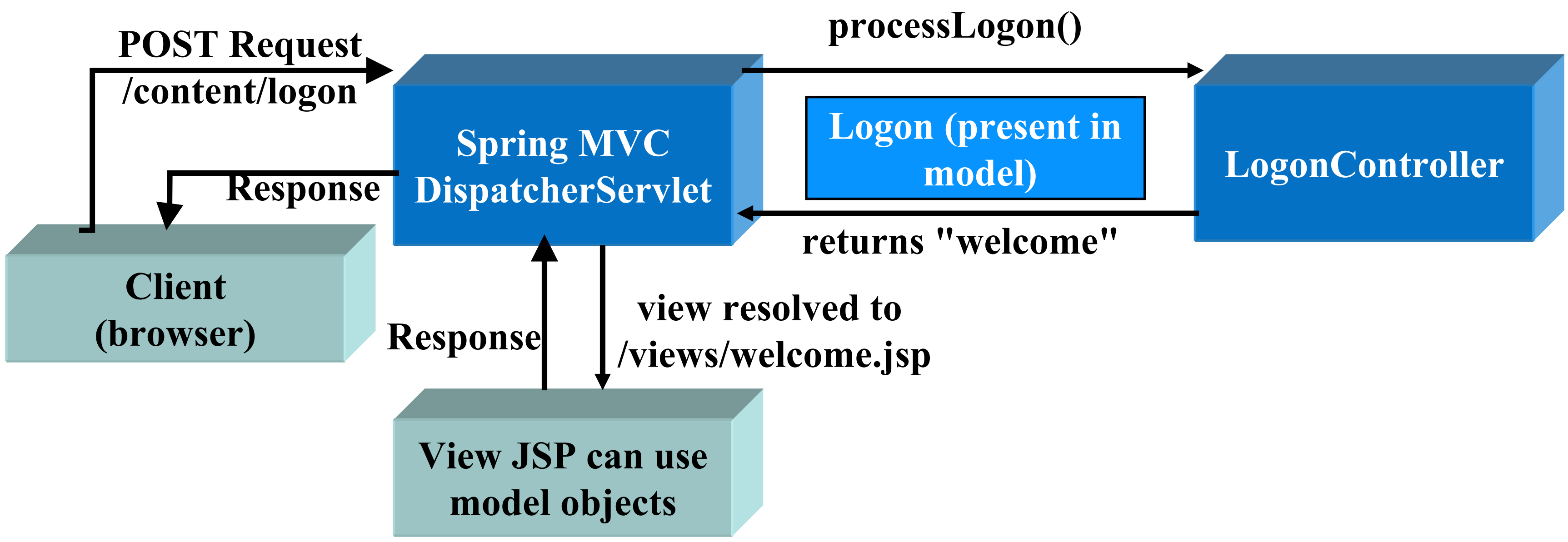
Controller Data Binding - @ModelAttribute

- `@ModelAttribute` binds incoming data to a model class
  - The model object's fields are populated by request parameters
  - Don't need to bind individual form fields
- Below, `processLogon()` has a model attribute (of type Logon)
  - It's `name` and `password` fields are bound to the incoming data from the form

```
@RequestMapping(method = RequestMethod.GET) // Generates page
public String get(@ModelAttribute("logon") Logon l) {
  l.setName("Jane Doe");  // Initialize model with default name
  return "logon";  // Return the view name
}

@RequestMapping(method = RequestMethod.POST) // Processes form
public String processLogon(@ModelAttribute("logon") Logon l) {
  // Assume checkLogon defined elsewhere - detail not shown
  l.setPrincipal( checkLogon(l.getName(),l.getPassword()) );
  return "welcome";
}
```

Request Handling Flow

- We illustrate the flow for processing a logon below
  - Note: The principal object is now a property in the `logon` bean
  - Before the principal basically was the model




```
<!-- welcome.jsp page - previously used ${principal.name} -->
Welcome ${logon.principal.name}
```

Reference Data with @ModelAttribute

- Models can be populated with reference data
  - As opposed to request data
- **@ModelAttribute** methods are called before request processing
  - **Before** `@RequestMapping` methods in the same controller class
  - Can add non-request data into the model
- The example adds a model attribute `previousNames` to the model
  - Containing a list of previous logon names
  - You could add a `previousNames` property to `Logon` to hold this data

```
@ModelAttribute("previousNames")
public Collection<String> populateLogonNames() {
  ArrayList<String> names = new ArrayList<String>();
  names.add("");
  names.add("James Gosling");
  names.add("Jimmy G");
  return names;
}
```

Using Reference Data

- At bottom, we use the `previousNames` attribute that was added in
  - It populates a select holding a list of previous logon names
  - The select is generated by a `form:select`
    - Its **`items`** attribute specifies the model property used to populate the select

```
<form:form modelAttribute='logon'>
  <form:input size='20' path='name'/>
  <form:input size='20' path='password'/>
  <form:select path="previousName" items="${previousNames}"/>
  <input type='submit' name='Submit' value='Logon'/>
</form>
```

Lab 8.1: Forms and Model Objects

We work with Spring's form tags and with model objects

Working with Sessions

Forms and Model Objects

Working with Sessions

Validation

Storing Model Objects in the Session

- **@SessionAttributes** triggers model attribute storage in a session
  – The named model attributes are automatically stored in the session
  – Below, our Logon instance is stored in the session
  – Can provide single attribute name, or array of names

```
@Controller
@SessionAttributes("logon")
@RequestMapping("/logon")
public class LogonController {

  @ModelAttribute("logon")  // Create Logon - stored in session
  public Logon createLogon { return new Logon(); }

  @RequestMapping(method = RequestMethod.POST)
  public String processLogon(@ModelAttribute("logon") Logon l) {
    // Logon instance automatically retrieved from the session
    // Do whatever work you need
    return "welcome";
  }
}
```

Using Model Objects in the Session

- `doWork()` just uses the Logon object as usual
  - In the example below, we illustrate use of a `ModelMap`
  - We also could have used `@ModelAttribute` on a method arg [1]

```
@Controller
@SessionAttributes("logon") // It will be stored in the session
@RequestMapping("/work")
public class SensitiveWorkController {

  @RequestMapping(method = RequestMethod.POST)
  public String doWork(ModelMap model) {
    Logon l = (Logon)model.get("logon");
    Principal p = l.getPrincipal();
    // Check permissions, do some work, etc. …
    return "continue";
  }
}
```

SessionStatus - Clearing the Session

- The session attributes can be cleared using `SessionStatus`
  - In package `org.springframework.web.bind.support`
  - Calling `setComplete()`, as Illustrated below, triggers session cleanup
    - Removes session attributes added via `@SessionAttributes`

```
@Controller
@RequestMapping("/logoff")
public class LogoffController {

  @RequestMapping(method = RequestMethod.GET)
  public String finish(SessionStatus status) {
    status.setComplete(); // Removes "logon" object from session
    // Do any other needed processing
  }
}
```

Obtaining a Session Object

- May sometimes want to work with session directly
  - e.g. to access items directly from the session
  - Accessed via `HttpSession` parameter in a handler (see below)
- Assume we have a `Principal` instance in the session
  - Put there by a non Spring MVC filter
  - e.g. to check permissions for some action
  - It can be used by other handlers, and by non-Spring MVC
  - Has to be managed (e.g. removed) directly via the session API

```
@Controller @RequestMapping("/work")
public class SensitiveWorkController {

  @RequestMapping(method = RequestMethod.POST)
  public String doWork(HttpSession session) {
    Principal p = (Principal) session.getAttribute("principal");
    // Check permissions, do some work, etc. …
    return "continue";
  }
}
```

Lab 8.2: Working with Sessions

We work with the HTTP session from controllers. You will also finish a cart controller with substantial Spring MVC functionality

Validation

Forms and Model Objects

Working with Sessions

Validation

Validation Overview

- Spring MVC supports standard Java validation (JSR-303)
  - Uses annotations to add validation constraints
  - The simple example below states that the `name` and `password` properties must not be null
- Validation is automatically enabled if a validator implementation is on the classpath
  - e.g. Hibernate validator

```
import javax.validation.constraints.NotNull;

package com.javatunes.web;

public class Logon {
  @NotNull private String name;
  @NotNull private String password;

  // get/set methods not shown
}
```

Validation Constraints

- The validation API provides the following constraints
  - `@Null`: Must be null
  - `@NotNull`: Must not be null
  - `@AssertTrue`: Must be true
  - `@AssertFalse`: Must be false
  - `@Min (long value)`: Must have value >= the minimum
  - `@Max (long value)`: Must have value <= the maximum
  - `@DecimalMin (String value)`: Must have value >= the minimum
  - `@DecimalMax (String value)`: Must have value <= the maximum
  - `@Size (int min, int max)`: Must have a value between the limits
  - `@Digits (int integer, int fraction)`: Must be a number within the range
  - `@Past`: Must be a date in the past
  - `@Future`: Must be a date in the future
  - `@Pattern (String regexpr, Flag[] flags)`: Must match the specified regular expression (Flags offer regular expression settings)

Validation Usage

- Below, we illustrate a previous `LogonController`
  - Binds the name/password request parameters to a model object

- The validation checks will automatically run on the model object
  - An exception will be thrown if name or password is null

```
@RequestMapping(method = RequestMethod.POST) // Processes form
public String processLogon(@ModelAttribute("logon") Logon l) {
  // Remaining detail not shown
  return "welcome";
}
```

Manual Validation Configuration

- Configured automatically when Spring MVC enabled
  - Either via `@Configuration` or in XML
- `LocalValidatorFactoryBean` can be used to configure explicitly (shown using XML below)
  - Does all needed JSR-303 initialization
  - Supports injecting of a `Validator` into a bean
  - Can also inject a `ValidatorFactory` if you want to create the validator yourself
- Requires that a JSR-303 provider be present on the classpath
  - This will be detected automatically and used

```
<bean id="validator" class=
"org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"
/>
```

Programmatic Validation

- Can be done with a `javax.validation.Validator`
  - Generally injected, as shown below
  - Below, if validation fails a `ValidationException` is thrown when `validate()` is called

```
import javax.validation.Validator;

public class UserService {

    @Autowired
    private Validator validator;

    public void checkLogon (Logon l) {
      validator.validate(l);
      // Other code omitted ...
    }
}
```

Review Questions

- How do controllers help you process forms?
  - What role to Spring model classes play in this?
- How do controllers access HTTP sessions, and how is session data use?
- What support does Spring provide for validation?

Lesson Summary

- Controllers can easily process form requests
  - They can bind method parameters to model classes to gain access to the form data
  - `@ModelAttribute` is an annotation that does this
- Controller methods can access an HTTP session by including one in its parameter list
  - It will automatically be initialized with the current session
  - `@SessionAttributes` indicates to the runtime that a model attribute should be stored in the session
- Spring MVC supports using JSR-303 style validation
  - It can easily be integrated with the MVC container

LEARNINGPATTERNS

Session 9: RESTful Services with Spring

REST Overview and Principles

Requests and Responses

REST with Spring MVC

Ajax Overview

Lesson Objectives

- Understand what REST is and its fundamental principles
  - Review / strengthen understanding of HTTP
  - REST is **all about HTTP** – and that means **all of HTTP**
  - Understand a REST API and how to invoke a RESTful service
- Learn the RESTful services API in Spring
  - Build and deploy Spring-based REST resources
- Access REST resources via Ajax

REST Overview and Principles

REST Overview and Principles

Requests and Responses

REST with Spring MVC

Ajax Overview

NOTE: Our Domain Model

- We will illustrate our REST resources, with several JavaTunes entities (listed below)
  - Self-explanatory, and should be clear from the examples
  - `item`: An item available in the store
    - As seen previously
  - `customer`: A customer of JavaTunes
    - Customers place orders for items
  - `order`: A request by a customer to buy items

REST – Representational State Transfer

- Set of **principles** for defining and accessing network ***resources***
  - – No official REST specification
    - BUT you'll need the HTTP specification handy! [RFC 2616]
  - – We'll describe generally accepted principles

- REST focuses on **resources** identified by **URIs**
  - – **Data centric**, not API centric
  - – Interactions are **stateless** between requests
  - – Resources have a **simple, uniform interface**

- Radically different from SOAP – and **much simpler**
  - – Detailed comparison later

REST Characteristics

- REST services are **stateless**, and parties communicate by **transferring representations of resources**
  - Every request carries the information needed to carry it out
  - **Simple representations** (XML, JSON, text)
- **Interaction is simple – only 4 methods**
  - GET, POST, PUT, DELETE
- **Built on existing protocols and standards**
  - HTTP used <u>exclusively</u>
  - Universal data representations (XML, JSON, text)
- REST is the **<u>full actualization of HTTP</u>** (1)
  - Understanding HTTP = understanding REST

REST Resources and Operations

- REST resources are **uniquely identified by URI**
  - The URI is a **noun** – the resource being accessed

- Operations on resources are defined via **HTTP methods**
  - The method is a **verb**

- **GET retrieve** resource's representation
- **POST create** new resource
- **PUT update** existing resource
- **DELETE delete** resource

- What exactly **is** a "resource?" It's any business entity
  - Customer, Order, Product, Forecast, Stock, Student, Article, JobPost, Bank, Loan, Account, etc.

REST in the Real World

- Twitter
  - *http://dev.twitter.com/rest/public*
- Facebook
  - *http://developers.facebook.com/docs/graph-api*
- LinkedIn
  - *http://developer.linkedin.com/docs/rest-api*
- Reddit
  - *http://reddit.com/dev/api*
- Instagram
  - *http://instagram.com/developer*
- Weather Underground
  - *http://wunderground.com/weather/api/d/docs*

Yes, Google uses it, too

REST Principles

- **Resources and addressability**
  - Everything is a resource (1)
    - Let's look at a resource for maintaining orders (e.g. for javatunes)
  - Every resource has a unique URI

/orders	all orders, or create new order
/orders/237	a specific order
/orders/237/items	all items for a specific order
/orders/237/items/14	a specific item in a specific order

- **Uniform constrained interface**
  - Only 4 operations available – GET, POST, PUT, DELETE (2)
  - But wait: isn't that limiting me to simple "CRUD" operations?
    - **No** – this is the "consistent foundation API" – you can accomplish everything you need to, using these 4 operations
    - Sometimes requires an adjustment in thinking (see notes)

REST Principles

- **Uniform set of resource representations**
  - "Objects," called ***entities***, are transmitted as XML or JSON
    - Format expressed via standard MIME types, e.g., application/json
  - Sent in the body of the HTTP request / response

- **Uniform set of headers and status codes**
  - HTTP has a rich set of request and response headers
    - Accept and Content-Type, plus many others
  - And a well-understood set of response status codes
    - 200 OK, 201 Created, 404 Not Found, 500 Internal Server Error, etc.

- These principles make REST **familiar** – and thus popular
  - Unless you've never used the Web...
    - Even your mother probably knows what a 404 means(!)

Requests and Responses

REST Overview and Principles

Requests and Responses

REST with Spring MVC

Ajax Overview

REST APIs

- Set of URIs + associated verbs, their effects, and responses
- Consider our orders resource and its methods and effects

  /orders

  - **GET** request returns a **list of all orders**
    - In XML or JSON – dictated by service implementation and client preferences – a process called ***content negotiation*** (more later)
  - **POST** request creates a **new order**
    - Includes the order information in the request body (in XML or JSON)
- Same URI here, differentiated by HTTP verb (method)
- In contrast, a SOAP-based service defines several operations unique to the service – all invoked via POST
  - `<getCustomer>`, `<listAll>`, `<remove>`, `<update>`, etc.

URI Templates

- URIs with embedded variables, denoted in **{ }**
  /orders/**{id}**

- Clients substitute values to produce a concrete URI
  /orders/**237**

- Flexibly identify single-valued resource, e.g., a specific order

- Provide **structural description** of a service interface
  - Describing the possible URI forms for the service
    /orders
    /orders/**{orderId}**/orders/**237**
    /orders/**{orderId}**/items /orders/**237**/items
    /orders/**{orderId}**/items/**{itemId}** /orders/**237**/items/**14**

REST API with URI Template

- Supported operations for **/orders/{id}** might include:
  - **GET** retrieves that specific order
    - No request body
    - 200 OK + response body (XML or JSON) →

      or
    - 404 Not Found

```
<order id="237">
  ...
  <total>27.48</total>
</order>
```

  - **DELETE** deletes that specific order
    - No request body
    - 204 No Content + no response body [indicates successful delete]

      or
    - 404 Not Found
  - **PUT** updates that specific order
    - Request body contains updated order info →
    - 204 No Content + no response body

      or
    - 404 Not Found

```
<order id="237">
  ...
  <total>24.73</total>
</order>
```

GET – Retrieve All Items

```
GET /items HTTP/1.1
Accept: application/xml, text/xml
```

Accept = format I want back

```
- no body -
```

```
HTTP/1.1 200 OK
Content-Type: application/xml
```

Content-Type = format of body

```
<items>
  <item id="456">
    <title>Ride the Lightning</title>
    <artist>Metallica</artist>
    <releaseDate>1982-03-02</releaseDate>
    <price>14.99</price>
  </item>
  <item id="457">
    <title>Bake Sale</title>
    <artist>Sebadoh</artist>
    <releaseDate>1994-08-23</releaseDate>
    <price>12.49</price>
  </item>
  ...
</items>
```

GET – Retrieve Specific Item

```
GET /items/322 HTTP/1.1
Accept: application/xml, text/xml
```

```
- no body -
```

```
HTTP/1.1 200 OK
Content-Type: application/xml
```

```
<item id="322">
  <title>The Colour and the Shape</title>
  <artist>Foo Fighters</artist>
  <releaseDate>1997-05-20</releaseDate>
  <price>16.97</price>
</item>
```

OR

```
HTTP/1.1 404 Not Found
```

```
- no body -
```

POST – Create New Item

```
POST /items HTTP/1.1
Content-Type: application/json
```

```
{
  "title" : "Lovedrive",
  "artist" : "Scorpions",
  "releaseDate" : "1979-02-28",
  "price" : 12.49
}
```

```
HTTP/1.1 201 Created
Location: /items/655
```

```
- no body -
```

- Response's **Location** header indicates URI of new resource
  - Typically, ID is generated on the server
  - If client wants it, sends a GET request to this new URI

PUT – Update Existing Item

```
PUT /items/322 HTTP/1.1
Content-Type: application/xml

<item id="322">
  <title>The Colour and the Shape</title>
  <artist>Foo Fighters</artist>
  <releaseDate>1997-05-20</releaseDate>
  <price>12.97</price>
</item>
```

was 16.97

```
HTTP/1.1 204 No Content

- no body -
```

- 204 No Content indicates a successful operation
  - Client already has current data for this resource, so no response entity needed
  - If a response entity is included, then use 200 OK

DELETE – Delete Existing Item

```
DELETE /items/322 HTTP/1.1
```

```
- no body -
```

```
HTTP/1.1 204 No Content
```

```
- no body -
```

OR

```
HTTP/1.1 404 Not Found
```

```
- no body -
```

- 204 No Content indicates a successful operation
- 404 Not Found is used for id not found
  - Including subsequent calls to the same DELETE URI

Some Additional Points

- ***Safe*** methods
  - GET and HEAD are read-only and thereby "safe"
- ***Idempotent*** methods
  - The effect of multiple identical requests is the same as just one
    - **"Effect" = the state of the system (on the server)**
  - GETs do not change system state
  - Multiple identical PUTs has same effect as a single one
  - Same with DELETEs
  - **POST** is the **only** method that is **not** idempotent
- Other HTTP methods
  - HEAD: identical to GET but never any response body
  - OPTIONS: provides info about a resource

Summary of REST Characteristics

- **Not a standard – leverages existing standards**
- **Simple representations** (XML, JSON)
- **Small number of fixed methods** (GET, POST, PUT, DELETE)
  - With many different resources – identified by different URIs
- **Simple interactions**
  - Request can be as simple as GET /customers/123
  - Response is a simple "customer" document (XML or JSON)
- **Fully leverages HTTP** – all of it
  - Methods, entities, headers, response codes, caching, etc.
  - Authentication, privacy, intermediate proxies

REST in Action – Mini-Lab

- Easy to test drive REST services – with no coding
  - Myriad tools exist to help you create and send HTTP requests, including the common everyday browser!

- Later, we'll write Ajax-JavaScript clients and Java clients

Mini-Lab

- Directions and weather – pretty common desires...
  - And available via several public RESTful services

- On the filesystem, navigate to ***workspace/MiniLab09.1***

  see notes

  - Follow instructions in the text files – do Google Maps first
  - Try out the examples, experiment with your own
  - These are all GET requests – you can just use a browser
    - NOTE: IE may not render JSON directly, recommend another browser

REST with Spring MVC

REST Overview and Principles

Requests and Responses

REST with Spring MVC

Ajax Overview

Spring Support for REST

- REST support builds on top of Spring MVC, including:
  - **`@RequestMapping`** maps RESTful URIs to standard controllers
  - **`@PathVariable`** supports URI templates
  - **`@RequestBody`** binds incoming HTTP data to Java objects
  - **`@ResponseBody`** binds return values to the response body
  - **HTTP Method conversion** supports GET, POST, DELETE, PUT
  - **New views** support XML and JSON response representations
  - We'll go into detail on most of these

- Generally, you map the **Spring MVC Dispatcher servlet** to a URL specifically for REST requests
  - e.g. ***/rest/****
  - All such requests would be dispatched via the dispatcher servlet

A Simple Resource Class Example

- **CustomerResource** is a resource mapping to requests of the form

  *<webapp>/rest/**customers***

  – GET request to ***/customers*** lands on `getAllCustomers()`

  – GET request to ***/customers/123*** lands on `getCustomer()`

    • The 123 value is bound to the `Long id` parameter

- `@RequestMapping` works as seen previously for Spring MVC

  – Let's look at the URI Template

```
@Controller
@RequestMapping("/customers")
public class CustomerResource {

  @RequestMapping  // Find/return all customers
  public String getAllCustomers() { … }

  @RequestMapping("/{id}") // Find/return a single customer
  public String getCustomer(@PathVariable Long id) { … }
}
```

URI Templates and @PathVariable

- Below the URI for `getCustomer()` is **/customers/{id}**
  - It has a URI Template with a variable named **`id`**
  - It matches URIs like **/customers/123**

- **`@PathVariable`** binds the `id` method parameter to this variable
  - Based on matching the variable name to the parameter name
  - **`123`** is bound to the **`id`** method parameter in this example

```
@Controller
@RequestMapping("/customers")
public class CustomerResource {

  @RequestMapping("/{id}")
  @ResponseBody // We'll cover this next
  public String getCustomer(@PathVariable Long id) { /* … */ }
}
```

Generating a Response - @ResponseBody

- **`@ResponseBody`** indicates the return value contains the response itself
  - It is not a view name or model object [(1)]

- Return data is **automatically converted** to a representation the client can accept
  - One that satisfies the client request's **`Accept` header**
  - e.g. a **`text/xml`** accept header results in **XML** being returned
  - If there's no Accept header, any representation may be used
  - Spring provides **Http message converters** for this conversion
    - It ships with a number of different converters that will be automatically applied as needed

@ResponseBody Example

- Below, **@ResponseBody** is applied to `getCustomer()`
  - The string value of a customer is returned (using `toString()`)
    - The customer is found based on the id from the URI template
- Here, a simple string representation is used as the response body
  - To get us started
  - We'll show other representations later

```
@RequestMapping("/{id}")
@ResponseBody
public String getCustomer(@PathVariable Long id) {
  Customer result = … // Find a Customer (not shown)
  return result.toString();
}
```

Other URI Path Mapping Capabilities

- **Multiple URI Templates** in a URI
  - `/customers/fname/{fname}/lname/{lname}` (see bottom)
  - `/customers/{fname}/{lname}`
    - Both URI Templates above have two variables
    - The first example includes literal path elements (1)
- **Specification of variable names** in `@PathVariable`
  - When parameter/variable names don't match
  - Below, we specify the use of `fname` to initialize `firstName` and `lname` for `lastName`

```
@RequestMapping("/fname/{fname}/lname/{lname}")
public String getCustomer(
      @PathVariable("fname") String firstName,
      @PathVariable("lname") String lastName)
{ /* Detail omitted */}
```

@RestController

- **@RestController** combines @Controller and @ResonseBody
  - Eliminates need for @ResponseBody on resource methods
  - Useful when a class only provides REST data (1)
  - In package org.springframework.web.bind.annotation

```
@RestController
@RequestMapping("/customers")
public class CustomerResource {

  @RequestMapping("/{id}")  // @ResponseBody not needed
  public String getCustomer(@PathVariable Long id) {
    Customer result = … // Find a Customer (not shown)
    return result.toString();
  }
}
```

But How Does It Work?

- That's the beauty of it – it just works!

- Seriously, the Spring MVC/REST **implementation** drives it
  - **Maps incoming HTTP requests** to the proper resource methods
  - **Reads individual HTTP request values**
    - Presents them to your resource method, as input arguments
  - **Reads XML and JSON entities** from the request body
    - Converting them into Java objects
  - **Writes return values** into XML or JSON entities in the response body
  - And much more

```
<item id="322"> <-> Item object
```

- Implementations also use functionality from the web container
  - REST resources are packaged / deployed as a standard WAR

Accessing REST Resources

- Consider a resource to manage customer orders
  - URI **/orders**
  - URL http://javatunes.com / **store** / **rest** / **orders**

webapp context path rest-path resource URI

/store webapp context root

/rest base path for all REST resources in the application

/orders a REST resource

- Generally refer to resources by using just the **resource URI**

- GET request to **/orders** returns a representation of all orders
  - In XML or JSON

Lab 9.1: A Simple REST Resource

Create a RESTful Resource Using Spring MVC

Ajax Overview

REST Overview and Principles

Requests and Responses

REST with Spring MVC

Ajax Overview

Ajax-JavaScript and REST Resources

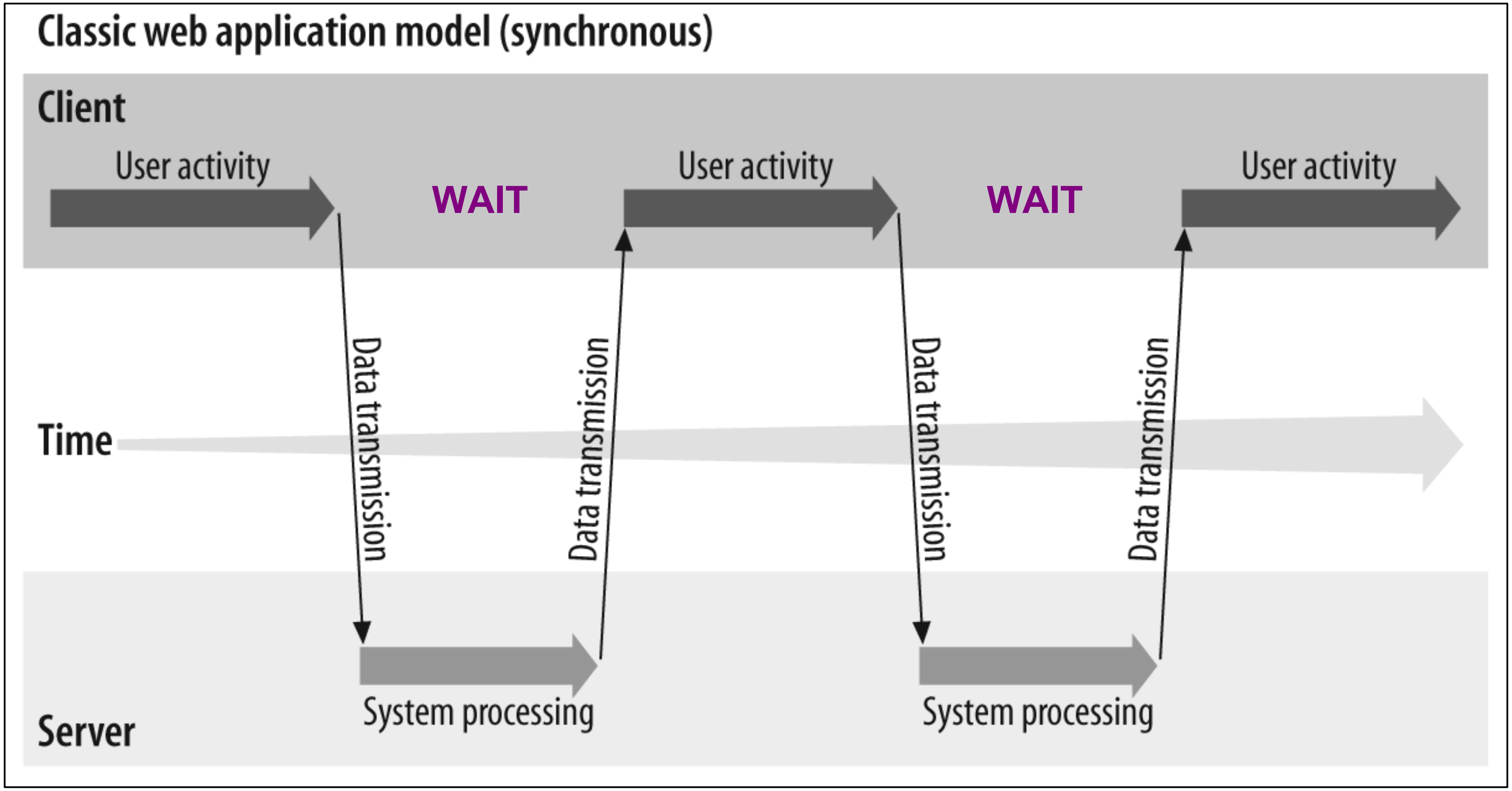
- Browsers are a common REST client
  - Often use JavaScript with **Ajax**-based interaction

- **Ajax = Asynchronous JavaScript and XML**
  - Technique for creating responsive and interactive web apps
    - Clients **exchange small amounts of data** with the server
    - **Only the affected parts** of a page are updated
    - Entire page does not need to be refreshed

- In page-oriented apps, **each request** generates a **new page**
  - Workflow / user interaction based on pages
  - User waits for server response
  - Slower, less interactive, and less intuitive

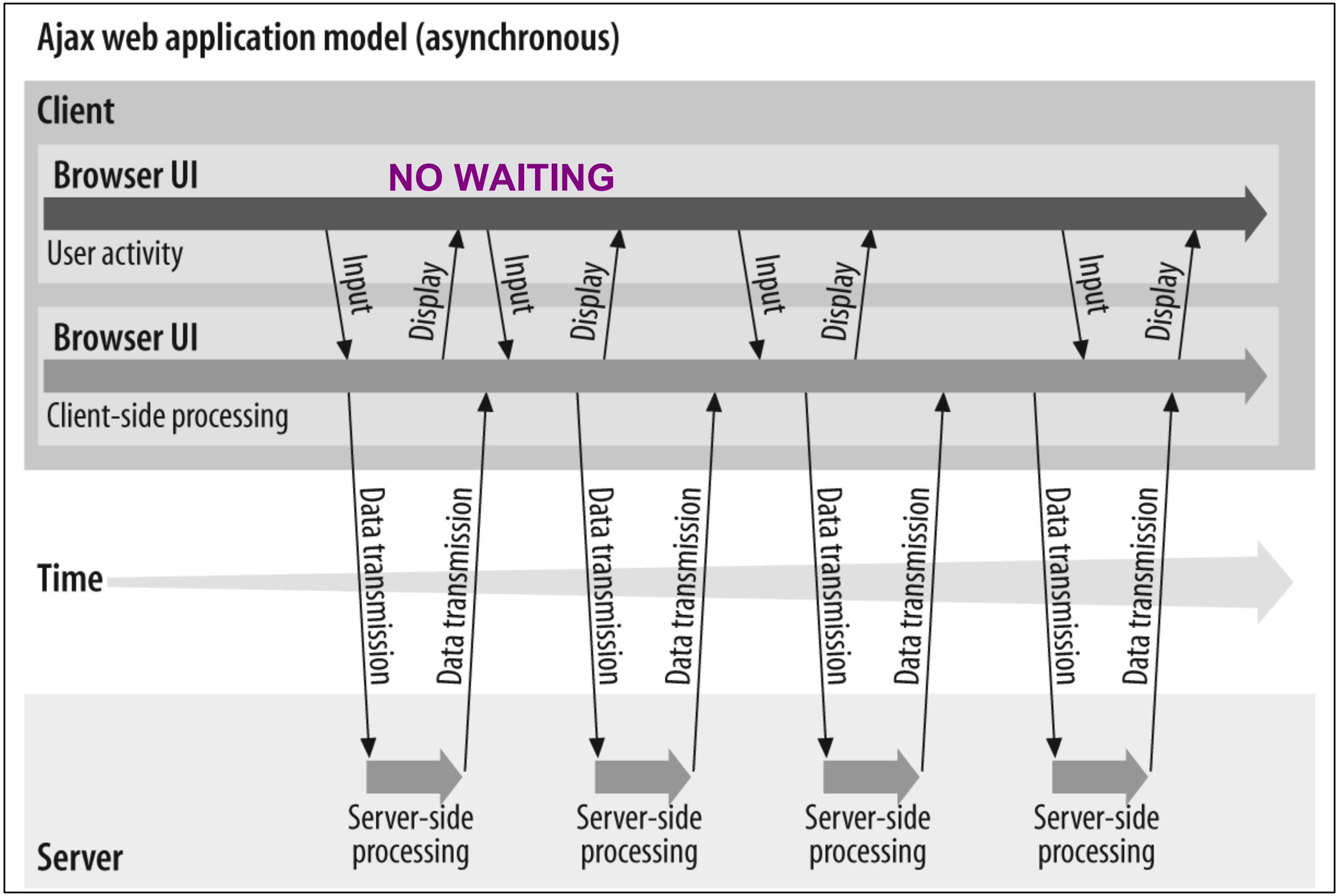
Classic User Interaction – Synchronous

- User makes request, **waits** for complete page refresh
  – Over the years, web pages have become "larger" and more complex, compounding the problem



Ajax User Interaction – Asynchronous

- Response fetched in background, user continues to work
  - Pervasive in modern web applications



Working with Ajax-JavaScript

- Ajax can be done in the browser via direct JavaScript code
  - Using the browser's **`XMLHttpRequest`** object
- But typically, high-level JavaScript libraries are used instead
  - We'll use **jQuery** in this course
    - The Ajax-JavaScript code is provided in the labs
    - No need to know Ajax or JavaScript
    - But you may need to modify the code to correctly access your RESTful resources
- jQuery is a popular JavaScript library and framework for development of dynamic web apps
  - Great toolkit for JavaScript and Ajax development
  - See notes for a reference list of several others

Ajax and REST Resources

- Below, jQuery is used to access a REST resource
  - Queries for a single customer via the **REST resource URI**
    - **`/customers/123`** (customer with `id == 123`)
  - Refreshes part of the page with the REST data
    - We'll do this in the lab also
  - jQuery details are not significant - this is for illustration only [(1)]
  - We provide all needed jQuery code

```
<script>
  $( document ).ready(function() {
    $.get('/myapp/rest/customers/123', function(responseData) {
      $('#ajaxContent').html(responseData);
    });
  });
</script>
```

Lab 9.2: Use Ajax in a Client

Access REST data from an Ajax client (a Web page)

Review Questions

- What is REST? What are RESTful services?
- How are Spring REST and Spring MVC related?
- How do you write a REST resource using Spring MVC/REST
- What is Ajax, and why do we care about it?

Lesson Summary

- **REST** is a data-centric set of principles for networked resources.
  - Resources are identified by URI, and have a simple uniform interface built on HTTP methods (GET, POST, PUT, DELETE)
  - REST interactions are stateless between requests.
- Spring REST builds on top of Spring MVC controllers:
  - `@RequestMapping` maps RESTful URIs to standard controllers
  - `@PathVariable` supports URI templates
  - `@RequestBody` binds incoming HTTP data to Java objects
  - `@ResponseBody` binds return values to the response body
  - HTTP Message converters support different data representations
- Ajax - Technique for creating responsive and interactive web apps.
  - Clients (browsers) get small amounts of data from a server, and only update affected parts of a page.
  - A full page refresh is not needed.

Session 10: Working with JSON and XML

Generating JSON

Generating XML

Content Negotiation

Generating JSON

Generating JSON

Generating XML

Content Negotiation

JSON Overview

- **JSON** (JavaScript Object Notation) is a lightweight data interchange format
  - Based on the JavaScript object/array structure
- JSON is a simple, easily understandable text format
  - Very familiar to anyone who has worked with JavaScript
  - Often, the client side is JavaScript in a browser
- Very common data format
  - Browser/tool support is excellent
  - Supported in many server-side technologies
  - Often used in Ajax/REST-based systems

JSON Details

- JSON is based on JavaScript object / array structure
  - A JSON value is one of:
    - **number**
    - **string** (in double quotes)
    - **object** (in `{ }`)
    - **array** (in `[ ]`)
    - `false` `true` `null`

- This object has three attributes
  - **description** (string), **x** (number), y (number)

```
{
    "description": "a line with two points"
    "x":2,
    "y":4
};
```

Arrays and More Complex Objects

- Arrays contain zero or more elements enclosed in square brackets ([ ])
  - The elements may be simple values, objects, or other arrays

- This has two attributes: **description** (string), **points** (array)
  - Each object in the array has two attributes: **x** and **y** (number)

```
// Object with a string property and an array property
{
  "description" : "A line with two points" ,
  "points" : [
     {"x":5, "y":6},
     {"x":6,"y":7}
   ]
}
```

JSON Representations for REST Resources

- REST data is generally more complex than a string
  - e.g., a Java object, as shown at bottom
- **Return object converted to JSON** if the client requests it
  - e.g. request includes **`application/json`** in the **Accept header**
  - Done via Spring MVC **HTTP message converters**
- Below, `getCustomer()` returns a customer object
  - Which can be converted to JSON when appropriate

```
@RequestMapping("/{id}")
@ResponseBody
public Customer getCustomer(@PathVariable Long id) {
  Customer result = … // Find a Customer (not shown)
  return result;
}
```

Http Message Converters / JSON

- Spring ships with several message converters
  - Used to convert to/from a representation
  - e.g. `Customer` object to JSON string

- **`MappingJackson2HttpMessageConverter`**
  - Jackson-based JSON converter that ships with Spring
  - Reads/writes JSON using Jackson 2.x's `ObjectMapper`
  - **Jackson**: popular open-source JSON/XML framework.
  - The converter can automatically produce JSON in resource methods
    - By converting Java objects to JSON
  - In `org.springframework.http.converter.json`

Summary: JSON Serialization Requirements

Server

- Make sure a JSON library is on the classpath
  - e.g. the Jackson libraries
- Make sure converters are registered
  - This is already true when using Spring MVC/REST
    - Done by `mvc:annotation-driven/@EnableWebMvc` (1)
- Return an appropriate data object from the REST resource
  - Which can be converted to JSON

Client

- Request/consume the JSON data
  - Send a request that **`accepts application/json`**
  - Process it (e.g. with jQuery - we'll demo this in the lab)

Lab 10.1: A JSON Resource

Create a RESTful Resource Returning JSON Data

Generating XML

Generating JSON

Generating XML

Content Negotiation

XML Review

- XML: **e<u>X</u>tensible <u>M</u>arkup <u>L</u>anguage**
  - A **meta-markup language** for representing **data**
  - XML **documents** contain markup (mostly **tags**) and data
  - Both markup and data are in plain text
- Below, we show a sample XML customer document
  - Assuming a customer held all that data

```xml
<?xml version='1.0' encoding='UTF-8' standalone='no'?>

  <customer ID='1'>
    <name>Jane Doe</name>
    <street>1475 Cedar Avenue</street>
    <city>Fargo</city>
    <state>ND</state>       <!-- must use abbreviation -->
    <zipcode>58103</zipcode>
    <shipper name='FedEx' accountNum='893-192'/>
  </customer>
```

Introducing JAXB

- **JAXB = Java Architecture for XML Binding**
  - Translate between Java object and XML representation
    - ***Marshalling*** write Java object to XML
    - ***Unmarshalling*** read XML into Java object
  - Included in JavaSE, in the **`javax.xml.bind`** packages

also called "serializing" and "deserializing"

- **Required** for XML support in RESTful services

- Its annotations will be your main involvement

see notes

  - Declare a class as "JAXB-capable"
  - Customize the nature of the XML

- Other classes you may encounter:
  - `JAXBElement`, `JAXBContext`, `Marshaller`, `Unmarshaller`

`@XmlRootElement` and Default Mapping

- Remember the fundamental idea:
  - **JSON or XML across the wire**
  - **Java objects in and out of your resource methods**
- For XML support, classes must be **"JAXB-capable"**
  - Annotated with **`@XmlRootElement`**
  - Generally called ***JAXB classes***
- Default mapping is **all public members**
  - Every public getter / setter pair and every public field
  - The XML element names also take defaults
- This is the same default mapping as for JSON (so far)

Default JAXB Mapping – Example

- The child elements of **<person>** are there because of the **get/set methods**
  - <u>NOT</u> because of the fields

```
@XmlRootElement
public class Person {
  private String name;
  private Integer age;
  private Boolean isMarried;
  // get/set methods for each
  // written in same field order
}
```

↔

```
<person>
  <age>50</name>
  <married>true</married>
  <name>Jason</name>
</person>
```

- Can you see how the default mapping works?

- Technically, the default order is "undefined"
  - **But** the JAXB runtime actually does it **alphabetically** (see notes)

Serializing Data to XML

- Uses Spring's message converters - two choices for XML
- **`Jaxb2RootElementHttpMessageConverter`**
  – Registered if **JAXB libraries on classpath** (present in Java 6+)
  – Reads and writes XML via JAXB2
  – Supports `text/xml` or `application/xml`
  – Supports all JAXB annotated objects (`@XmlRootElement`)
- **MappingJackson2XmlHttpMessageConverter**
  – Registered if **Jackson 2.1+** and its **XML extension** on classpath
  – Reads/writes all JAXB annotated classes
  – Additionally writes non-JAXB annotated classes and collections
  – Supports `text/xml`, `application/xml`, `application/*+xml`
  – More convenient than JAXB - we'll demo in lab

Using @XmlRootElement

- **JAXB** (Java Architecture for XML Binding): Defines mapping between XML documents and Java objects
  - Part of its functionality defines annotations like `XmlRootElement` that describe how Java classes map to XML documents
  - More detail is beyond the scope of the course

- `javax.xml.bind.annotation.`**`XmlRootElement`**
  - JAXB annotation for annotating Java types
  - Indicate type maps to an XML element
  - **Required if using JAXB** to convert top-level class to XML (1)
    - Not required by Jackson converter

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Customer { /* Detail omitted */ }
```

Serializing Collections to XML

- JAXB can't directly serialize a Java collection
  - `java.util` collections aren't JAXB annotated objects
  - They must be wrapped in a JAXB-annotated object
  - Jackson doesn't require the wrapper (1)
  - We'll use a wrapper in the lab

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name="customers")
public class CustomerCollectionWrapper {
 private Collection<Customer> collection;
 public CustomerCollectionWrapper(Collection<Customer> colIn) {
  collection = colIn;
 }
 public Collection<Customer> getCustomers() {return collection;}
 public void setCustomers(Collection<Customer> colIn) {collection = colIn;}
}
```

```
  @RequestMapping  // Find/return all customers
  public CustomerCollectionWrapper getAllCustomers() {
    Collection<Customer> col = // … Get this somehow
    return new CustomerCollectionWrapper(col);
  }
```

Summary: XML Serialization Requirements

Server

- Make sure an XML serialization library is on the classpath
  - JAXB will always be there - it's in the JDK now
  - Or use the Jackson libraries with its XML extension
- Make sure converters are registered
  - This should be the case anyway if using Spring MVC/REST
- Return an appropriate data object from the REST resource
  - If using JAXB one that is annotated with `@XmlRootElement`
  - So it can be converted to XML

Client

- Request/consume the XML data
  - Send a request accepting XML - e.g. **`application/xml`**
  - Process it (e.g. with jQuery - we'll demo this in the lab)

Content Negotiation

Generating JSON

Generating XML

Content Negotiation

Content Negotiation Overview

- Client and resource method must **agree on content type**
  - Client sends JSON, resource method `consumes` JSON
  - Client wants XML back, resource method `produces` XML

- Either or both parties may be willing to "negotiate"
  - Client is more often "single-minded" here
    - I send JSON, and I want JSON back – period
  - Resource method should be the more flexible one
    - I can do JSON or XML – have it your way
    - **Best Practice**: Resource methods **support both** - **in** and **out**

- Spring's **content negotiation** determines final data format
  - Takes into account **what client asks for** and **resource methods can produce**

Accept Headers: What the Client Asks For

- Spring examines HTTP's **Accept header**
  - To determine what a client can accept
  - Clients (browsers and others) can set this
  - Frameworks (like jQuery) can set it directly
    - e.g. to say "I accept **JSON** only"
  - Browser's default accept header is complex and confusing (1)
    - Generally accepting HTML, and possibly many other things

- Spring supports other conventions for accept
  - **Path extension** (suffix) on a URL, e.g. to specify XML
    http://foo.com/crm/rest/customers.**xml**
  - **URL parameter** (by default called "format"), e.g.
    http://foo.com/crm/rest/customers**?format=xml**

Determining Client's Data Format

- Spring checks for client's capabilities in this order:
  - Path extension (suffix)
  - URL parameter - **disabled** by default
  - Accept header
  - Default content type

- You can customize the content negotiation via configuration
  - We show an example below
  - See the documentation for more detail (1)

```
@Configuration @EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
 @Override
 public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
    configurer.favorPathExtension(false).
            favorParameter(true).
            ignoreAcceptHeader(true).
            defaultContentType(MediaType.APPLICATION_JSON);
  }
}
```

Configuring Controller's Data Types

- Controllers can configure both the data type they consume and produce
  - Using `@RequestMapping` elements, as shown at bottom
    - **`produces`**: List of mime types produced
    - **`consumes`**: List of mime types accepted
  - Spring considers these when mapping a request to a controller
    - e.g. a request for JSON must go to a controller producing JSON

```
// This produces only JSON, so we no longer have to wrap it (1)
@RequestMapping(produces="application/json")
public Collection<MusicItem> getAllCustomers() { … }

// This produces both XML and JSON (2)
@RequestMapping(value="/{id}",
                produces={"application/xml", "application/json"})
public Customer getCustomer(@PathVariable Long id) { … }
```

Lab 10.2: An XML Resource

Create a RESTful Resource Returning XML Data

Review Questions

- What is JSON, and why do we care about it?
- How do we indicate that an entity is XML "capable"?
- How does content negotiation work?

Lesson Summary

- JSON: Common lightweight data interchange format
  - Based on the JavaScript object/array structure
- XML "capable" entities are generally annotated with @XmlRooElment (JAXB)
  - Or an equivalent - e.g. one of the Jackson annotations that it supports
  - The structure of the XML can also be defined using these annotations
- In content negotiation, the client and the resource method agree on content type
  - Since both often work with different content types, the "negotiation" is based on what the client can accept and the resource produce

Session 11: Java Clients for RESTful Services

Learn how to create RESTful clients with Spring's RestTemplate

NOTE: If Java-based clients are not of interest, you can skip all of this session, or do up to the first lab for reduced coverage

REST Client Requirements

- General requirements for any REST client:
  - **Create a connection** to the server
  - **Set request headers** (e.g. Accept headers)
  - **Add request data** (URI Template variables / request params)
    - e.g. start with a base URI of *http://foo.com/crm/rest/customers*
    - Then add id data to the base - e.g. by concatenating a string
  - **Invoke** the REST service
  - **Extract** any results, and possibly convert them

- Spring's **`RestTemplate`** takes care of much of this
  - Similarly to other templates, e.g. database access
  - Much easier than coding it yourself
  - Requires Spring jars on the client

RestTemplate Example

- At bottom, `RestTemplate` accesses a customer
  - Retrieving by id via **`RestTemplate.getForObject`**
  - The URI Template (`{id}`) is expanded from the value `"123"`
  - More detail shortly
- With this single call, `RestTemplate` handles
  - Connecting to the server
  - Adding the URI Template to the URI
  - Converting to a Java object (generally from XML/JSON)
  - It supports all HTTP methods (GET, POST, PUT …)

```
String URI = "http://foo.com/crm/rest/customers/{id}";
RestTemplate rt = new RestTemplate();
Customer found = rt.getForObject(URI, Customer.class, "123");
```

Common RestTemplate Methods

- Below, are some common `RestTemplate` methods
  - Note the naming pattern (HttpMethod-for-ReturnValue) (1)
  - We'll go into more detail on these shortly
    - As well as more detail on the REST patterns used

HTTP Method	RestTemplate method	Description
DELETE	delete()	Delete resources at given URI
GET	getForObject(), getForEntity()	GET as object or `ResponseEntity`
HEAD	headForHeaders()	Retrieve headers
OPTIONS	optionsForAllow()	Retrieve an Allow header
POST	postForLocation(), postForObject()	POST for location of resulting object, or for actual object
PUT	put()	Create new resource via PUT
PATCH, others	exchange(), execute()	See notes (2)

getForObject() in Detail

- **`getForObject()`**: Performs a **GET** for a Java **object**
  - Hence the name `getForObject`
  - We illustrate below (1 of 3 available variations):

```
public <T> T getForObject(
   String url,                      // The request URL
   Class<T> responseType,           // Response type
   Object... urlVariables)          // URI Template values
 throws RestClientException
```

  - The **`url`** can include URI Template Variables
  - **`responseType`** is type of return object
    - Converted via HTTP message converters - same as for the service
  - **`urlVariables`** are for expanding URI Template Variables
    - A **varArgs** value - see notes

RestTemplate Method Variations

- `RestTemplate` has multiple methods for each HTTP method
  - Convenience methods for different situations
  - e.g. the `getForObject()` variations below

```
T getForObject(URI url, Class<T> responseType)
```

  - Doesn't support URL Template expansion

```
T getForObject(String url, Class<T> responseType,
                      Object... urlVariables)
```

  - **`urlVariables`**: **varArgs** list of values to expand URL template vars
  - Values are assigned by their order in the url

```
T getForObject(String url, Class<T> responseType,
                      Map<String,?> urlVariables)
```

  - **`urlVariables`**: **Map** of values to expand URL templates vars
  - Consist of key (string) / value pairs - values are keyed by name

getForObject() Using Map

- Below, we illustrate a URI Template with two variables
  - Uses a **map** to supply the values
    - Map is name-based
    - Does not depend on the order of variables or values
    - Useful for multiple variables

```
RestTemplate rt = new RestTemplate();
String URI =
 "http://foo.com/crm/rest/customers/fname/{fname}/lname/{lname}";
Map<String,String> varsMap = new HashMap<String,String>();
varsMap.put("fname", "Jane");
varsMap.put("lname", "Doe");
Customer found = rt.getForObject(URI, Customer.class, varsMap);
```

Other RestTemplate Methods

```
void delete(String url, Object... urlVariables)
```

- Delete the resources at the specified URI

```
T postForObject(String url, Object request,
   Class<T> responseType, Object... uriVariables)
```

- Create a new resource by POSTing the given object to the URI

```
void put(String url, Object request,
         Object... urlVariables)
```

- Create or update resource by PUTting the given object to the URI

–We'll cover some of these other methods in more detail later

Lab 11.1: A Client Using RestTemplate

In this lab, we will access a RESTful resource from a standalone Java client using `RestTemplate`

RestTemplate.getForEntity()

- **getForEntity()** performs a GET request for an HTTP entity
  - Returns **ResponseEntity** (extends **HttpEntity**)
    - Represents a request or response entity
    - Allows access to HTTP specific data of a response
    - In package `org.springframework.http`

- `getForEntity()` signature (1 of 3 variations):

```
<T> ResponseEntity<T> getForEntity(
   String url,                  // The request URL
   Class<T> responseType,       // Response type
   Object... urlVariables)      // URI Template values
 throws RestClientException
```

  - The arguments are as described earlier
  - Usage on the next page

Using HttpEntity and ResponseEntity

- **`HttpEntity`** / **`ResponseEntity`** represent a request or response entity
- Contains headers (in an `HttpHeaders` object) and a body
  - Access the headers and body via the methods below

```
public HttpHeaders getHeaders()
public T getBody()
```

  - `ResponseEntity` adds the method `getStatusCode()`
    - As well as methods to build a response on the server side
  - Below is a simple example

```
String URI = "http://foo.com/crm/rest/customers/{id}";
RestTemplate rt = new RestTemplate();
ResponseEnity<Customer> foundEntity =
    rt.getForEntity(URI, Customer.class, "123");
System.out.println(foundEntity.getStatusCode());
System.out.println(foundEntity.getBody().getName());
```

Accessing Headers

- **`HttpHeaders`** wraps the header data
  - **Common headers** have specific access methods, e.g.

    `long` **`getContentLength`**`()`: Content-length header value

    `MediaType` **`getContentType`**`()`: Content-type header value

  - **Generic access** methods are useable for any header, e.g.

    `String` **`getFirst`**`(String headerName)`: First value for header

  - Many others - see the documentation
  - Below is a simple example of accessing headers

```
    // Assume same URI and RestTemplate object as before …

ResponseEnity<Customer> foundEntity =
    rt.getForEntity(URI, Customer.class, "123");
System.out.println("getForEntity returns content type: " +
                        foundEntity.getHeaders().getContentType());
```

RestTemplate.exchange()

- `RestTemplate.exchange()` takes an `HttpEntity` object
  - Can be used to set headers for the request

```
<T> ResponseEntity<T> exchange(
   String         url,          // The request URL
   HttpMethod     method,              // HTTP method
   HttpEntity<?>        requestEntity, // Request
  entity
   Class<T>       responseType,
   Object...      urlVariables)
 throws RestClientException
```

  - `url`, `responseType`, `urlVariables` as seen earlier
  - **`method`** is the HTTP method to use - GET, POST, etc.
  - **`requestEntity`** is the entity to write to the request
    - May include headers

Setting Headers on a Request

- Below, `exchange()` sets an accept header specifying JSON
  - Using the enum `org.springframework.http.`**`MediaType`**
  - Lengthy, but straightforward
  - See notes for API on some of these types

```
// Assume same URI and RestTemplate object as before …

List<MediaType> accepts = new ArrayList<MediaType>();
accepts.add(MediaType.APPLICATION_JSON);

HttpHeaders headers = new HttpHeaders();
headers.setAccept(accepts);
HttpEntity<Customer> requestEntity =
        new HttpEntity<Customer>(headers);

ResponseEntity<Customer> exchangeEnt = rt.exchange(ID_URI,
        HttpMethod.GET, requestEntity, Customer.class,"123");
System.out.println(exchangeEnt.getHeaders().getContentType());
```

Error Handling

- Several choices for handling server errors
- e.g. if lookup by id fails, you can :
  - Throw an exception (e.g. `IllegalArgumentException`)
  - Return null
    - Both **result in a runtime exception** on the client
  - Provide for testing before a call
    - e.g. make sure that the id is found by the REST service shown below
- Client code can then choose an approach
  - e.g. handling exceptions, or calling a test method

```
// Based on REST controller used before

@RequestMapping(value="/exists/{id}", method=RequestMethod.GET)
@ResponseBody
public Boolean checkForCustomer(@PathVariable("id") Long id)
{ /* Check for customer with given id */ }
```

Summary of RestTemplate Usage

- We've looked at core RestTemplate functionality
  - Focusing on GET requests
- We've seen the many variants provided
  - Multiple versions of methods like `getForObject()`
  - Multiple methods for a particular task
    - e.g. `getForObject()`, `getForEntity()`, `exchange()`
- There is broad HTTP method support
  - Methods to do POST, PUT, DELETE, etc.
    - With multiple variants also
  - Other ways to execute requests (e.g. the `execute()` method)
- We'll cover some more of this later
  - There are many details, so look at the documentation !!

Lab 11.2: [Optional] Setting / Accessing Headers

In this lab, we will work with HTTP headers in both the request and response

Lesson Summary

- Spring's `RestTemplate` takes care of much of the boilerplate code for REST clients, including:
  - Handling connectivity requirements
  - Supporting creation of URIs, including URI Templates
  - Converting to/from Java objects to other representations e.g. XML or JSON
  - Supporting all HTTP methods (GET, POST, PUT …)

Session 12: Common REST Patterns

Explore common usage patterns for RESTful services

The REST Methods

- We'll review some standard REST methods (verbs)

- We'll look at these details for each method:
  - **REST method** (GET, POST, PUT and DELETE) and its effect
  - **CRUD equivalent operation**
    - Assuming a standard Create-Read-Update-Delete system
  - The **kind of entity** it operates on (the noun)
    - Either a single entity or a collection of entities
  - Other characteristics, e.g. **idempotent** or **cacheable**
    - Idempotent - can apply multiple times without changing the result
  - A **Controller implementation** example (the server side)
    - Without showing the business logic
  - A **`RestTemplate` client** example

GET: Retrieve Information

- **REST Verb**: GET (We've seen this already)
- **CRUD Equivalent**: Read
- **Noun**: Collection URI (multiple) or Entity URI (single)
- **Idempotent**: Should NOT initiate state change
- **Cacheable**
- **Examples** (only resource URI shown in all examples):
  - **Multiple**: GET /**customers**
  - **Single**: GET /**customers/123**

```
// Controller - Assumes @RequestMapping("/customers") on class

@RequestMapping // Multiple: Match URI like /customers
@ResponseBody
public CustomerCollectionWrapper getAll() {…}

@RequestMapping("/{id}") // Single: Match URI like /customers/123
@ResponseBody
public Customer getCustomer(@PathVariable Long id) { … }
```

POST: Add New Information

- **REST Verb**: POST
- **CRUD Equivalent**: Create
- **Noun**: Collection URI (Entity unknown before create)
- **Example**: POST /**customers**

- **Server**: Specify HTTP POST on the handler
  – Via `@RequestMapping's` **`method`** element (See next slide also)

```
// Controller - Assumes @RequestMapping("/customers") on class
// Must specify GET now (see notes)
@RequestMapping(method=RequestMethod.GET)
@ResponseBody
public Object getAll() {…}

@RequestMapping(method=RequestMethod.POST) // Matches on POST only
  @ResponseStatus(HttpStatus.CREATED)
  @ResponseBody
  public void createCustomer(@RequestBody Customer cust) {
    // Create a Customer from the passed in parameters
  }
```

Server: POST Controller - Details

- `@RequestMapping` specifies an HTTP POST
  - Via **`method=RequestMethod.POST`**
  - `getAll()` must now specify GET
    - Otherwise it handles all methods, producing a conflict
- `createCustomer()` receives a Customer object [1]
  - Created from the request body by an HTTP message converter
  - **`@RequestBody`** binds the `cust` method parameter to the request body
- Note the CREATED status code
  - We would return the created object in this example

```
@RequestMapping(method=RequestMethod.POST) // Matches on POST only
@ResponseStatus(HttpStatus.CREATED)
@ResponseBody
public Customer createCustomer(@RequestBody Customer cust)
{ /* … */ }
```

Server: Setting Location Header for POST

- POST controllers generally **set the location header**
  - Client can request a **location URI** instead of an object
  - And use the URI for later access

- We illustrate below
  - `createCustomer()` has an `HttpServletResponse` parameter
    - It's initialized with the current response object (see notes)
    - It's used to set the location header

```
@RequestMapping(method=RequestMethod.POST) // Matches on POST only
@ResponseStatus(HttpStatus.CREATED)
@ResponseBody
public Customer createCustomer(@RequestBody Customer cust,
                               HttpServletResponse response) {
  Customer created = // Create customer - detail not shown
  response.setHeader("Location", "/customers/" + created.getId());
  return created;
}
```

Client: RestTemplate for POST

- `RestTemplate.postXXX()` sends a POST
  - Creates a new object using data sent in the call
- `postForObject()` signature (1 of 3 variations):
  - POSTs to the specified URI with the given object

```
<T> T postForObject (
   String       url,                // The request URL
   Object       request,            // Object to create
   Class<T>     responseType)       // Type returned
 throws RestClientException
```

- Below is an example usage

```
String URI = "http://foo.com/crm/rest/customers";
RestTemplate rt = new RestTemplate();
Customer custData = new Customer("Jane", "Doe");
Customer created =
       rt.postForObject(URI, custData, Customer.class);
```

PUT: Update Information

- **REST Verb**: PUT
- **CRUD Equivalent**: Update
- **Noun**: Entity URI (URI of entity that is updated)
- **Example**: PUT /**customers/123**
  – URI includes the id - remaining data is in the request body
- **Server**: Specify PUT on the handler
  – `updateCustomer()` below, updates the passed in customer
  – Nothing returned here - can return updated object if you want

```
// Implementation - Assumes @RequestMapping("/customers") on class

  // Matches URI like /customers/123 with method of PUT (only)
  @RequestMapping(value="/{id}", method=RequestMethod.PUT)
  @ResponseBody
  public void updateCustomer(@PathVariable("id") Long id,
                             @RequestBody Customer cust) {
    // Update Customer with given id using values in Customer object
  }
```

Client: RestTemplate for PUT

- `RestTemplate.put()` sends a PUT
  - Must send the object to update in the call
- PUT method signature (1 of 3 variations):
  - PUT to the specified URI with the given object

```
<T> T put (
   String        url,                  // The request URL
   Object        request,              // Object to update
   Object...     urlVariables)         // URI Template values
 throws RestClientException
```

- Below, we update the customer retrieved for id = 123

```
String URI = "http://foo.com/crm/rest/customers/{id}";
RestTemplate rt = new RestTemplate();
Customer found = rt.getForObject(URI, Customer.class, "123");
found.setFirstName("NewName");
rt.put(URI, found, "123");
```

DELETE: Remove an Entity

- **REST Verb**: DELETE
- **CRUD Equivalent**: Delete
- **Noun**: Entity URI (URI of entity to delete)
- **Example**: DELETE /**customers/123**

- Server: Below we show a controller annotated for DELETE
  – Status is set to NO_CONTENT

```
// Implementation - Assumes @RequestMapping("/customers") on class

  // Matches URI like /customers/123 with method of DELETE (only)
  @RequestMapping(value="/{id}", method=RequestMethod.DELETE)
  @ResponseStatus(HttpStatus.NO_CONTENT)
  @ResponseBody
  public void deleteCustomer(@PathVariable("id") Long id) {
    // Delete customer with given id
  }
```

Client: RestTemplate for DELETE

- `RestTemplate.`**`delete()`** sends a DELETE

- `delete()` signature (1 of 3 variations):
  - Deletes the resource at the specified URI

```
public void delete (
    String url,                  // The request URL
    Object... urlVariables)      // URI Template values
  throws RestClientException
```

- Below, we delete the customer with id = 123

```
String URI = "http://foo.com/crm/rest/customers/{id}";
RestTemplate rt = new RestTemplate();
rt.delete(URI, "123");
```

[Optional] Lab 12.1: Additional REST Operations

Implement additional REST operations on the server side, and test them via a Java client

[Optional] Session 13: Additional New Features

Core Updates

WebFlux

Core Updates

Core Updates

WebFlux

Specification Baselines

- Spring 5 requires Java 8
  - Takes advantage of new features - e.g. interface default methods
  - And runs on Java 9
    - Using either classpath or module path
    - Passes JDK 9 test suite
    - Spring 5 framework build and test suite passes on JDK 9

- Requires JEE 7 compatibility in features (e.g. Spring MVC)
  - Servlet 3.1, JPA 2.1, JMS 2.0
  - Runs under JEE 8 (Servlet 4.0, JPA 2.2, etc.)

Lambdas for Bean Registration

- Java 8 lambdas support functional-style programming
  - Spring 5 now supports this style for bean definitions
  - Can use a lambda to register a bean
- `GenericApplicationContext` supports this via

```
registerBean(Class<T> beanClass)
registerBean(Class<T> beanClass,
        Supplier<T> supplier)
```

  - `java.util.function.Supplier` represents a supplier of T results

- We illustrate usage below - only likely to be used if your app is written in a functional style

```
GenericApplicationContext ctx = new GenericApplicationContext();
ctx.registerBean(InMemoryItemRepository.class);
// Use a lambda for the supplier
ctx.registerBean(CatalogImpl.class,
       ()->new CatalogImpl(ctx.getBean(ItemRepository.class)));
```

Default Interface Methods

- Many Spring interfaces now have default methods
  - The previous adapter classes are often deprecated
    - Their default implementations can go in the interface now
- We've seen this in Spring MVC
  - **WebMvcConfigurer** is implemented directly
    - Instead of extending `WebMvcConfigurerAdapter`
  - Saves inheritance for when you need it

```
@Configuration @EnableWebMvc @ComponentScan("com.javatunes.web")
public class WebConfig extends WebMvcConfigurerAdapter { // Old way
}
```

```
@Configuration @EnableWebMvc @ComponentScan("com.javatunes.web")
public class WebConfig implements WebMvcConfigurer {} // New Way
```

Candidate Component Index

- A ***META-INF/spring.components*** file, if it exists is used to determine Spring components
  - Instead of using scanning
  - Contains list of component candidate names (one per line)

- To generate this index, include dependency **`spring-context-indexer`** in your POM
  - Then build the archive (e.g. with **`maven package`**)

- Improves startup time for systems with many classes
  - Reading the index is fast, and time is basically constant
    - As opposed to scanning many classes
    - Takes increasingly more time as system grows

@Nullable and @NonNull

- Spring version of standard Java (JSR 305) annotations for handling null values more safely
  - In **`org.springframework.lang`**
    - Don't confuse with validation annotations (e.g. `@NotNull`)
  - Usable by frameworks for runtime checks
  - Usable by tools for determining program correctness
    - e.g. For `@Nullable` below, tools can make sure you check for null

```
@Component("musicCatalog") // Declares bean - most detail omitted
public class CatalogImpl implements Catalog {
  @NonNull @Autowired // We know itemRepository should never be null
  private ItemRepository itemRepository;
}
```

```
  @RequestMapping(method = RequestMethod.POST)
  public ModelAndView processSearch(
       @Nullable @RequestParam("name") String keyword { // Keyword may be null
    if (keyword == null { … }   // So we need to check for it.
    else { … }
  }
```

LEARNINGPATTERNS

WebFlux (Reactive Systems)

Core Updates

WebFlux

Reactive Programming Overview

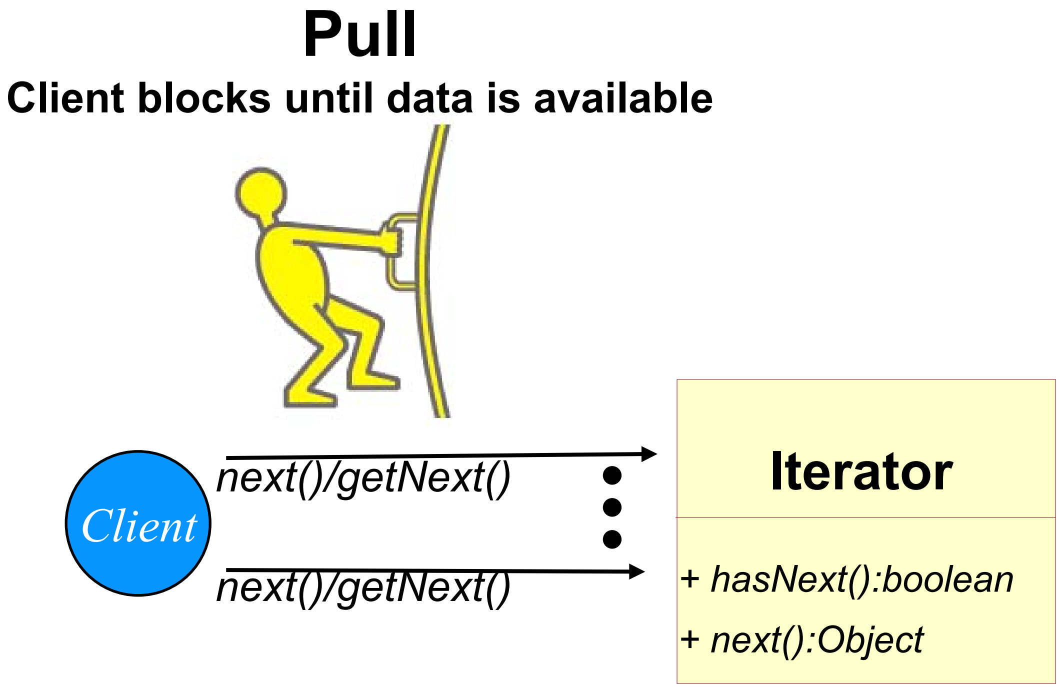
- **Reactive systems** are asynchronous and event driven
  - Require less resources (threads)
  - Scale better than synchronous systems

- Let's look at a system that processes multiple items
  - And assume we have **producers** and **consumers** of items
    - **Producer**: Source of the items
    - **Consumer**: User of the item

- Consider now, two core patterns for processing items
  - **Iterator**: The consumer asks the producer for each item
  - **Observer**: The consumer sends the producer each item
  - Let's look at them in detail
  - We'll see how one lends itself to reactive programming

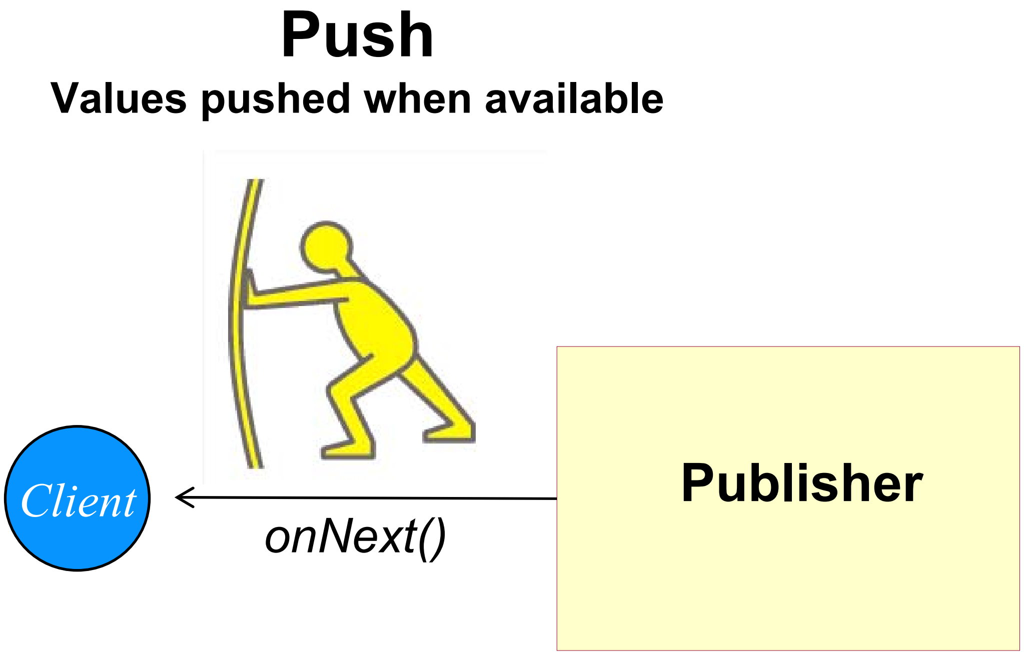
Iterator (Pull) - Synchronous

- **Iterator**: The **consumer asks the producer** for each item
  - The producer can notify the consumer that no items are left
  - Inherently **synchronous**
  - Consumer pulls items, and blocks until an item is available
  - When all items are consumed, the iteration finishes
  - Common model in previous generation web apps



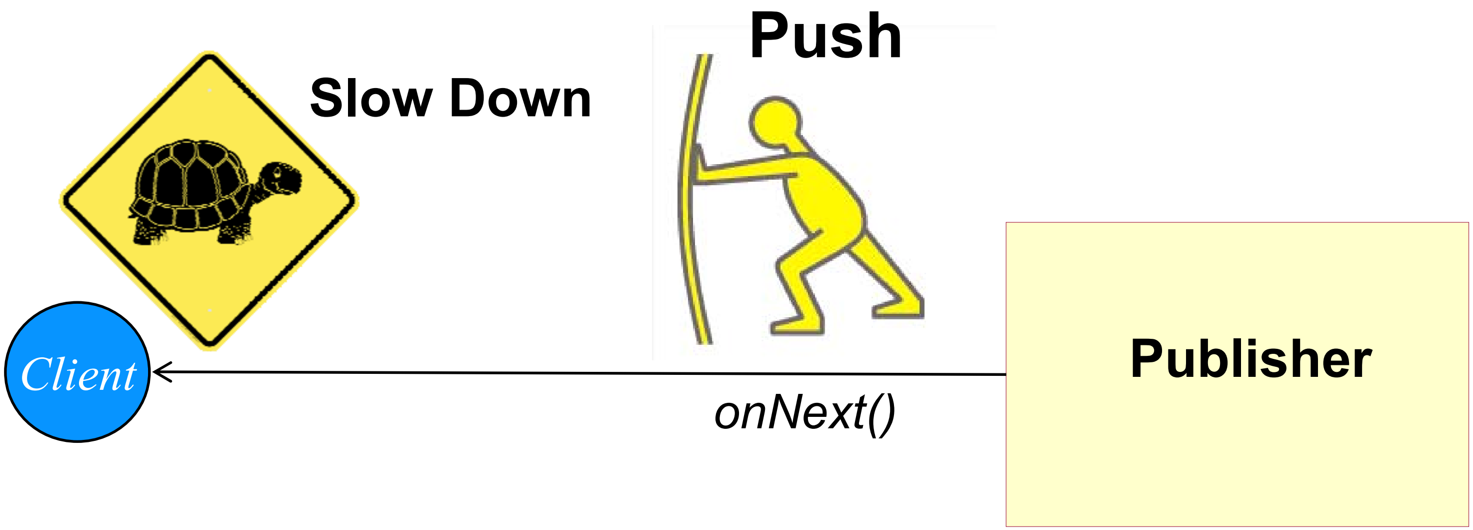
Observer (Push) - Asynchronous

- **Iterator**: The **producer sends the consumer** each item
  - e.g. based on an event occurrence
  - Inherently **asynchronous** - producer can push something at any time
  - Often call these producers "Publishers"



Reactive Programming - Observer+

- Reactive programming- observer with additional capabilities
  - Which makes Observer as capable as Iterator, but asynchronous!
  - Signal an **error** (`onError()` callback)
    - If error occurs, `onError()` called on consumer
  - Signal **completion** (`onCompleted()` callback)
    - When no more data available `onCompleted()` called on consumer
  - **Slow down** (back-pressure) - lets client perform load regulation
    - So it's not swamped



Spring Reactor / WebFlux Frameworks

- **Spring Reactor**: Core reactive API
  - **https://projectreactor.io/**
  - Conforms with **Reactive Streams** API (1)
  - Fully non-blocking reactive programming model
  - Integrates easily with Java 8 functional APIs (e.g. streams)
  - Large parts developed by Spring team, but separate from Spring
- **WebFlux**: Spring module supporting reactive programming
  - Reactive server Web apps (REST, HTML browser, WebSocket)
  - Reactive HTTP/WebSocket clients
  - Built on Reactor internally, and exposes Reactor API
  - Can use other Reactive frameworks (e.g. RXJava)

Note: Java 8 Lambda/Stream Usage

- The Reactor and WebFlux APIs were built to take advantage of Java 8 capabilities - in particular, the following:
  - **Lambda expressions**: Encapsulate a single unit of behavior
  - **Streams**: A series of operations on a sequence of elements
- We illustrate below
  - We get a stream from a collection (via Java 8's `stream()`)
  - We apply a filter operation on the items
    - Using a lambda for the filter
    - This lambda means "For the given item, evaluate `id%2==0`
  - More detail is beyond the course scope

```
ArrayList<MusicItem> items = // … Initialized somehow - not shown

// Use a stream to get all items with even id numbers
items.stream() // Get a stream
  .filter(item -> item.getId()%2==0) // filter the stream using a lambda
```

WebFLux API and REST Example

- We'll use two types in our examples
  - **Mono<T>**: Publisher that emits one result or an error
  - **Flux<T>**: Publisher that emits 0..N elements then completes
    - Successfully or with an error
  - More types/details are beyond scope of this overview
- Let's rewrite our familiar items resource in a reactive way
  - It's now written in terms of **Mono** and **Flux**
  - We illustrate the new method signatures below

```
@RestController @RequestMapping("/items")
public interface ItemsResource {

  @RequestMapping("/{id}")
  public Mono<MusicItem> findItem(@PathVariable("id")Long id) { /* … */ }

  @RequestMapping(produces = MediaType.APPLICATION_STREAM_JSON_VALUE) (1)
  public Flux<MusicItem> getAllItems() { /* … */ }
}
```

Example: Reactive Resource

- We illustrate the implementation below
  - Uses our previously seen `ItemRepository`
  - It's very simple - wraps the items in `Mono` or `Flux` as needed
  - Using convenient factory methods in the Reactor API

```
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono; // Other imports/detail omitted

@RestController @RequestMapping("/items")
public interface ItemsResource {
  @Autowired ItemRepository repo;

  @RequestMapping("/{id}")
  public Mono<MusicItem> findItem(@PathVariable("id")Long id) {
    return Mono.justOrEmpty(repo.findOne(id));
  }
  @RequestMapping(produces = MediaType.APPLICATION_STREAM_JSON_VALUE)
  public Flux<MusicItem> getAllItems() {
    Collection<MusicItem> items = repo.findAll();
    return Flux.fromIterable(items);
  }
}
```

Example: Reactive Client (Flux)

- Below, we illustrate a reactive client using Spring's **`WebClient`**
  - Converts response data to a **`Flux<MusicItem>`** and **filters** it
  - It subscribes to the `Flux` - supplying the processing as lambdas
  - Each element received is then processed asynchronously

```
// Much detail omitted - code fragment only.
String BASE_URI = "http://localhost:8080/javatunes/rest/items";
WebClient webClient = WebClient.create(BASE_URI);

Flux<MusicItem> foundAll = webClient.get()
        .retrieve()
        .bodyToFlux(MusicItem.class)
        .filter(item -> item.getId()%2==0);

 // This subscribe call takes three lambdas - what to do on success,
 // error, and completion respectively
 found.subscribe(
        successValue -> System.out.println(successValue),
          error -> System.out.println(error.getMessage()),
          () -> System.out.println("Mono consumed.")
        );
```

Example: Reactive Client (Mono)

- Below, is a similar client getting a single value
  - It converts the response data to a **`Mono<MusicItem>`**
  - It subscribes to the `Mono` similarly as for the `Flux`

```
// Much detail omitted - code fragment only.
String BASE_URI = "http://localhost:8080/javatunes/rest/items";
WebClient webClient = WebClient.create(BASE_URI);

Mono<MusicItem> found = webClient.get()
        .uri(ID_URI, "2")
        .retrieve()
        .bodyToMono(MusicItem.class);

 // This subscribe call takes three lambdas - what to do on success,
 // error, and completion respectively
 found.subscribe(
        successValue -> System.out.println(successValue),
          error -> System.out.println(error.getMessage()),
          () -> System.out.println("Mono consumed.")
        );
```

Recap

- **Con**: The code is more complex than previously
  - Can be a bit obscure, and harder to debug
  - Once you're used to lambdas and streams, it's relatively straightforward

- **Pro**: You get asynchronous systems
  - Which can be much more responsive, scalable, and flexible
  - This is a big win
  - Reactive systems are a major building block of modern systems
  - Spring WebFlux makes it very easy to write/consume reactive services

- We'll illustrate it at work in the demo lab

[Optional] Lab 13.1: WebFlux Demo

We illustrate a reactive resource and client written using WebFlux

[Optional] Session 14: XML Specific Configuration

Collection Valued Properties

Other Capabilities

These capabilities aren't relevant to @Configuration - where are usually straightforward (e.g. inheritance)

Lesson Objectives

- Review several configuration elements specific to XML
  - Generally, because they are not needed using `@Configuration`
  - We mention the `@Configuration` equivalents in the list below

- Initializing Collection Valued Properties
  - `@Configuration`: Work directly with collection class (e.g. `Set`)

- Bean definition inheritance
  - `@Configuration`: Inherit one configuration class from another

- Factory Classes
  - `@Configuration`: Use the factory class directly

Collection Valued Properties

Collection Valued Properties

Other Capabilities

Working with Collections

- Spring can configure properties containing a collection
  - Important collection configuration elements include:
  - **<list>**: A list of values, allowing duplicates
  - **<set>**: A set of values, allowing NO duplicates
  - **<map>**: A set of name-value pairs, where name and value can be of any type
  - **<props>**: A set of name-value pairs, where the name and value are both `Strings`
- `<list>` and `<set>` can be used for **array** valued properties or for implementations of `java.util.`**`Collection`**
  - The actual bean property does not have to be `List` or `Set`
  - The container enforces the list or set contract

Collection Property Example

- Consider the collection valued property shown at bottom
  - `InMemoryItemRepository` has a `catalogData` property
    - Defined as a `Collection<String>`
    - We use strings to simplify the example
  - The property can hold multiple music items
  - Let's configure this with XML

```
public class InMemoryItemRepository implements ItemRepository {
   // Much detail omitted …
   private Collection<String> catalogData;
   public void setCatalogData (Collection<String> catalogData) {
      this.catalogData=catalogData;
  }
   public Collection<String> getCatalogData() { return catalogData; }
}
```

Configuring <list> and <set> Properties

- Below, are two examples of initializing `catalogData`
  - `<list>`: Results in a collection with 3 values
  - `<set>`: Filters out the duplicate "`Abbey Road`" value
    - Results in a collection with 2 values

```
<bean id="catalog" class="com.javatunes.persistence.InMemoryItemRepository">
  <property name="catalogData">
    <list>
      <value>Abbey Road</value>
      <value>Tapestry</value>
      <value>Abbey Road</value>
    </list>
  </property>
</bean>
```

```
<property name="catalogData">  <!-- Most details omitted -->
  <set>
    <value>Abbey Road</value>
    <value>Tapestry</value>
    <value>Abbey Road</value>
  </set>
</property>
```

Configuring Collection of Bean References

- Collection properties may contain bean references
  - Configured similarly to value-based collections
  - Assume `InMemoryItemRepository` has a collection of type `MusicItem` (not just strings):

```
public void setCatalogData (Collection<MusicItem> data)
```

  - You could configure it as shown below

```
<!-- See notes for complete MusicItem bean def -->
<bean id="abbeyRoad" class="com.javatunes.domain.MusicItem">…</bean>
<bean id="tapestry" class="com.javatunes.domain.MusicItem">…</bean>

<bean id="catalog"
      class="com.javatunes.persistence.InMemoryItemRepository">
  <property name="catalogData">
    <set>
      <ref bean="abbeyRoad"/>
      <ref bean="tapestry"/>
    </set>
  </property>
</bean>
```

Map Valued Properties

- **Map** valued properties are also available
  - For example, if `InMemoryItemRepository` used a map - e.g.:

    ```
    public void setCatalogData (Map<String, MusicItem> data)
    ```

  - You could configure it using the configuration elements below

```
<bean id="catalog"
      class="com.javatunes.persistence.InMemoryItemRepository">

  <property name="catalogData">
    <map>
      <entry>
        <key><value>The Beatles</value></key>
        <ref bean="abbeyRoad"/>
      </entry>
      <entry>
        <key><value>Carole King</value></key>
        <ref bean="tapestry"/>
      </entry>
    </map>
  </property>

</bean>
```

LEARNINGPATTERNS

Other Capabililties

Collection Valued Properties

Other Capabilities

Bean Definition Inheritance

- A bean definition (**child**) can **inherit** configuration data from another bean (**parent**)
  - Useful to **share common configuration**
  - Possible to override some values or add values
- The **`parent`** attribute specifies a bean inherits configuration
  - Inherits the bean class, constructor-args, and all the properties
  - Can override all of these
  - Can merge collection of parent into that of child
  - Certain settings always taken from the child *
- An **`abstract="true"`** attribute is available for parent beans
  - The container won't instantiate abstract beans
  - Useful to define a bean ONLY to share configuration

Inheritance Example

- We illustrate inheritance and overriding at bottom
  - Assume `InMemoryItemRepository` had an integer property
    - `maxSearchResults`: Maximum number of results returned in one search (-1 means unlimited)
    - Below, the `catalog` bean sets a default value of 100
    - `unlimitedCatalog` overrides this to allow unlimited results

```
<bean id="catalog" class="com.javatunes.persistence.InMemoryItemRepository">
  <property name="catalogData">
      <set>
         <ref bean="abbeyRoad"/>
         <ref bean="tapestry"/>
      </set>
  </property>
  <property name="maxSearchResults" value="100"/>
</bean>

<!- Inherit definitions from parent - see notes for merge example -->
<bean id= "unlimitedCatalog" parent="catalog">
  <!-- Override this property: -->
  <property name="maxSearchResults" value="-1"/>
</bean>
```

Factory Classes

- Spring supports the creation of beans via factory methods
  - Let's look at **static factory methods**
- **`WarehouseFactory`**, shown below, defines **`getWarehouse()`** to create `Warehouse` instances
  - Rather than using `new Warehouse()`
- A <bean>'s **`<factory-method>`** attribute tells Spring to use a factory method, as shown at bottom

```
public class WarehouseFactory { // Details omitted
   public static Warehouse getWarehouse() { /* ... */ }
}
```

```
<bean id=“warehouse" class="com.javatunes.domain.WarehouseFactory"
      factory-method="getWarehouse">
  <!-- Most detail omitted -->
</bean>
```

Instance Factory Methods

- Spring also supports **instance factory methods**
  - Often encountered when using factory classes
- **`WarehouseFactory`**, shown below, now has an instance method to produce `Warehouses`
- `<bean>` uses a **`factory-bean`** attribute rather than `class`
  - The factory method is invoked on the specified factory bean **instance**

```
public class WarehouseFactory { // Details omitted
   public Warehouse getWarehouse() { /* ... */ }
}
```

```
<bean id="myFactory"
      class=com.javatunes.domain.WarehouseFactory"/>

<bean id="myWarehouse"
      factory-bean="myFactory" factory-method="getWarehouse">
</bean>
```

Autowiring with XML

- Spring can **autowire** dependencies via XML
  - We illustrate at bottom (uses a bean attribute)
  - The Spring container automatically injects dependencies into the `musicCatalog` bean (e.g. a repository) based on type matching
  - There are other capabilities (see notes)
- We've seen **better ways to do this**
  - This type of autowiring is fragile, and vulnerable to subtle bugs

```
<!-- Much detail / declarations omitted -->

   <bean id="inMemoryRepository“
      class="com.javatunes.persistence.InMemoryItemRepository"/>

   <bean autowire="byType" id="musicCatalog"
                           class="com.javatunes.service.CatalogImpl"/>
```

Inner Beans

- **inner bean**: Bean defined inside a `<property>` or `<constructor-arg>` element
  - Anonymous and don't need an id
  - Accessed through the containing bean, and not by name
  - Always scoped as prototype (No prototype attribute needed)
  - The containing bean owns the inner bean completely
- `catalog` below, uses an inner bean for its `itemRepository` property
  - It's accessed through catalog's `itemRepository` property
  - The `itemRepository` bean has no id declared

```
<bean id="catalog" class="com.javatunes.service.CatalogImpl">
 <property name="itemRepository">
  <bean class="com.javatunes.persistence.InMemoryItemRepository"/>
 </property>
</bean>
```

Compound Names

- **Compound property names** can be used to access nested beans
  - Consider `InMemoryItemRepository`
    - Assume it has a `maxSearchResults` property to limit the number of items returned
  - Below, **`itemRepository.maxSearchResults`** accesses the `maxSearchResults` property of `itemRepository`

```
<bean id="catalog" class="com.javatunes.service.CatalogImpl">
  <property name="itemRepository">
    <bean class="com.javatunes.persistence.InMemoryItemRepository"/>
  </property>
  <property name="itemRepository.maxSearchResults" value="99"/>
</bean>
```

Recap

Recap of what we've done

- We've covered a large part of the Spring Framework
  - The **core Spring 5 container** with bean configuration, Dependency Injection, resource access, bean lifecycle management, events, etc.
  - Spring Boot to simplify dependency management and configuration
  - **Database access** with Spring, including **Hibernate/JPA-based Repositories** and **Spring Data** for automatically generated queries
  - **Controlling transactions** declaratively with Spring using Spring annotations and XML configuration
  - Integrating Spring with **Web apps** using `ContextLoaderListener`
  - Using **Spring MVC** to build well-structured Spring Web applications
  - Using Spring MVC to create **RESTful resources**

What Else is There

- Spring is a large framework
  - We covered the structure and core ideas of all the areas we touched on
  - There is lots of opportunity for you to learn as you get more familiar with Spring
  - Read the documentation, read the books, use the Web
  - There is a lot of information out there to help you

- There are also additional capabilities that Spring provides
  - Look at the Spring reference manual - it has a lot of information on what the Spring capabilities are
  - There is much to explore in the wide Spring world

Resources

- The **Spring documentation**
  – The reference manual, API docs, and samples
  – They're all available in the full Spring download
- **http://spring.io/**
  – The home of the Spring Framework
- **Spring in Action** – Fifth Edition by Craig Walls [1]
  – An excellent book, full of useful examples and explanations
  – It's big - 600 pages +
- There are many more - these are a great place to start