LEARNINGPATTERNS

Presentation Slides:
For INSTRUCTOR Use Only

Labs: Introduction to Spring 5 and Spring MVC/REST (Eclipse/Tomcat)

Version: 20180521-b

Release Level

- This manual has been tested with, and contains instructions for, running the labs using the following platforms:
  - **Spring Boot** (tested with 2.0.0.RELEASE)
  - **Spring 5** (tested with 5.0.4.RELEASE)
    - Release used by Spring Boot
  - **Java** (tested with and requires Java 8)
  - **Eclipse Java EE Edition** (tested with Oxygen 4.7.2))
  - **Tomcat** for the Spring/Web material (tested with Tomcat 8.5)
- Recent similar versions of the software will likely work except for potentially small configuration changes

LEARNINGPATTERNS

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

Lab Synopsis

Lab

- **Overview**: In this lab, we will:
  - Become familiar with the lab structure
  - Set up our environment, including the Spring container
  - Write, compile and run a simple unit test that requires Spring and validates our setup
- **Builds on previous labs**: None
- **Approximate Time**: 30-45 minutes

Information Content and Task Content

Lab

- Within a lab, **information only** content appears the same as in the student manual pages
  - Like these bullets at the top of the page
- **Tasks to perform** are in an easily identifiable box
  - An example appears below

Tasks to Perform

- Note the different look of this instruction box compared to the above
  - Future labs will also use this format
- OK – Now **locate your setup files**; we're ready to start working (1)

Set Up Environment

Lab

- You'll need the lab setup file, with a name like
  - ***LabSetup_Spring5-MVC-REST_2018050521.zip***
- Our base working directory (created when we extract the setup) is: ***C:\StudentWork\Spring***
  - Assuming you extracted the zip to C:\ - **otherwise adjust accordingly**
  - It includes a directory structure and files needed for the labs

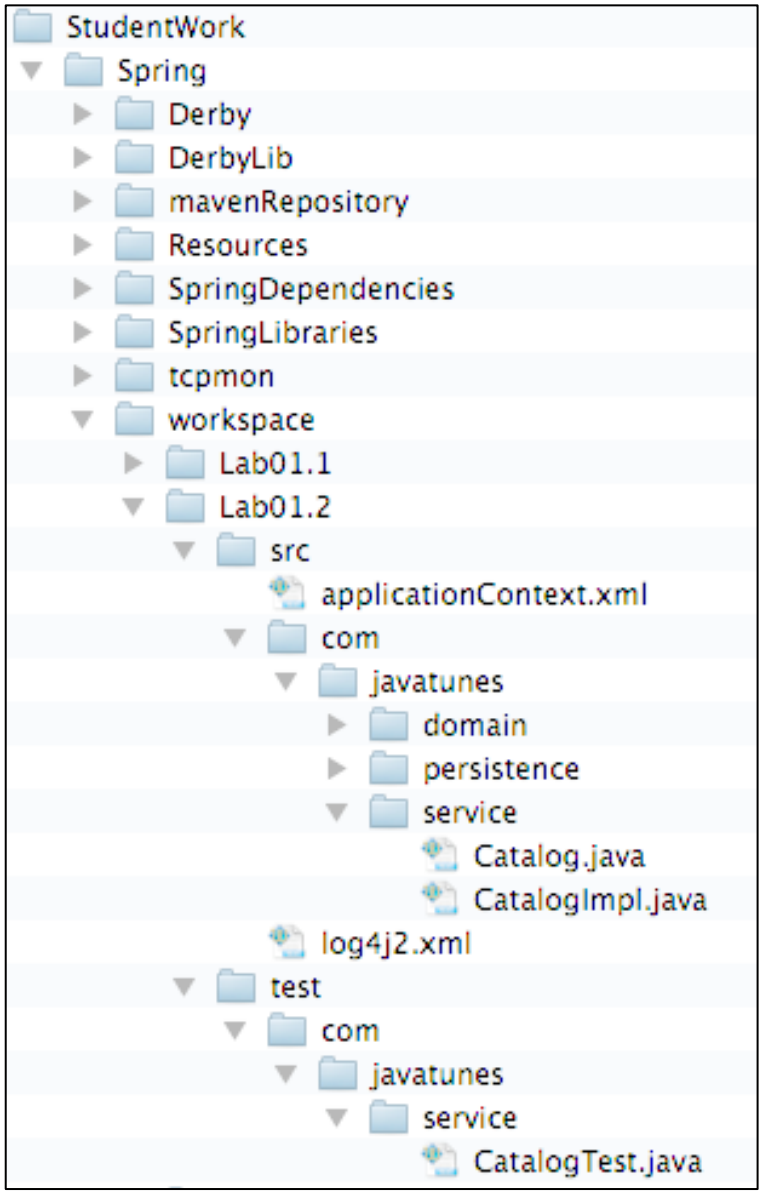
Tasks to Perform

- Make sure Java is installed
- Make sure Eclipse is installed
- Unzip the lab setup file to **`C:\`**
  - This will create the directory structure, described in the next slide, containing files that you will need for doing the labs
  - You can unzip it elsewhere - just adjust all locations consistently (1)

Lab Directory Structure

Lab

- **StudentWork\Spring** contains
  - **Derby/DerbyLib**: Database files
  - **mavenRepository**: pre-populated repo for labs
  - **SpringDependencies:** Dependency jars
  - **SpringLibraries:** Spring jars
  - **workspace**: Eclipse workspace
- **StudentWork\Spring\workspace** will contain the following folders:
  - **LabNN** : Folder for Lab NN
  - **LabNN/src** : Java source
  - **LabNN/test** : Java unit tests
  - **LabNN/bin/** : compiled code (Eclipse's standard location)



General Lab Structure

- Root lab folder for this lab, already in your workspace is:

  workspace\Lab01.1

  – It contains starter files
    - All labs either have starter files or build on a previous lab
  – You'll create an Eclipse project in this folder
  – Lab files and instructions are relative to the Lab01.1 folder

- Two source folders
  – ***src***: Contains regular Java source
  – ***test***: Contains JUnit test classes
    - To test our code
    - You'll run these using Eclipse's JUnit support
    - *test* may sometimes contain a regular Java app (with a `main()`)

Eclipse Overview

Lab

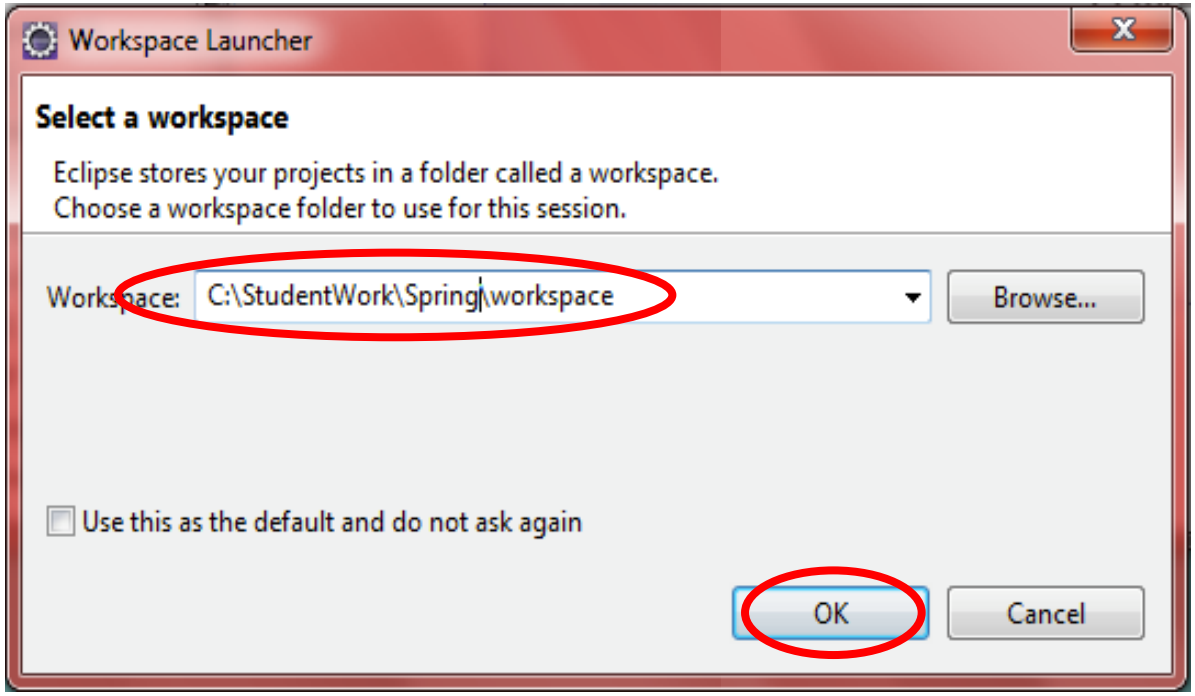
- **Eclipse**: Open source platform for building integrated development environments (IDEs)
  - Used mainly for Java development
  - Can be flexibly extended via plugins to add capabilities
  - **http://www.eclipse.org** is the main website
- This lab includes **detailed instructions** on using Eclipse
  - Starting it, creating and configuring projects, etc.
- Other labs include **fewer Eclipse details** - they may just say build/run as previously
  - Just use the same procedures to build/run as in this lab
  - Refer back to these lab instructions as needed

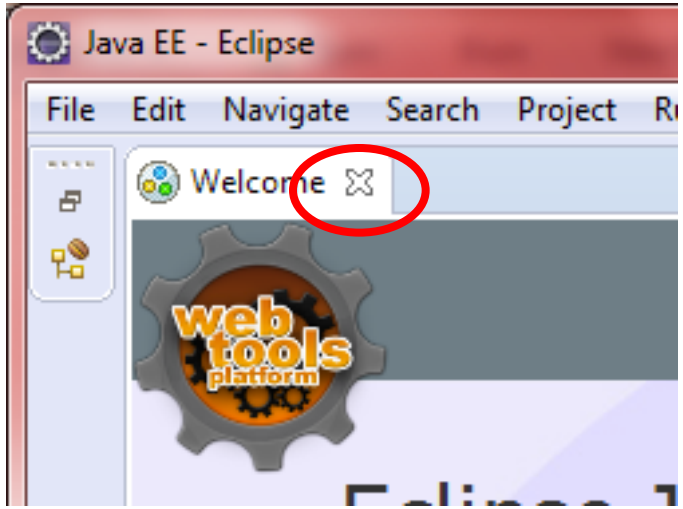
Launch Eclipse

Lab

Tasks to Perform

- Launch eclipse: Go to **c:\eclipse** and run **eclipse.exe**
  - A dialog box appears prompting for workbench location (below left)
  - Set the workbench location to **C:\StudentWork\Spring\workspace**
  - Click **OK**
  - **Close** the Welcome screen: Click the X on its tab (below right)



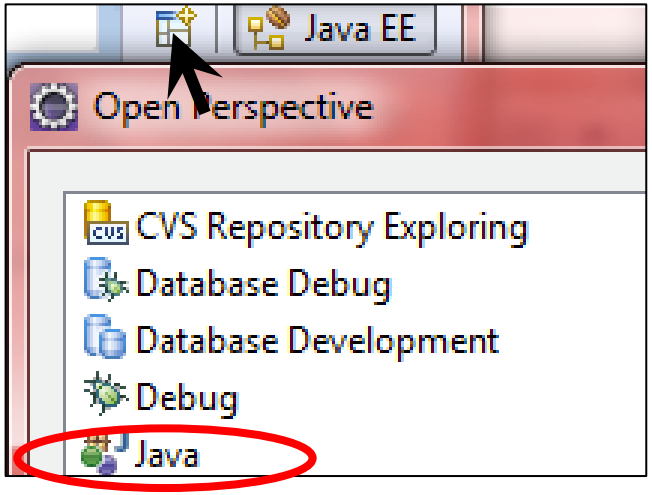


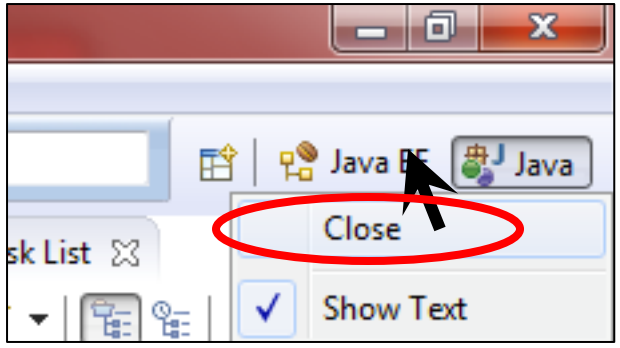
Workbench and Java Perspective

Lab

Tasks to Perform

- **Open a Java Perspective: Click the Perspective icon** at the top right of the Workbench, and select Java (bottom left)
  - See notes on Perspective Icons and text labels
  - Close the Java EE perspective by right clicking its icon, and selecting close (bottom right)
- Unclutter the Java Perspective by closing some views
  - **Close** the Task List and Outline views (click on the X)
  - Open the Navigator View (**Window | Show View | Navigator**)

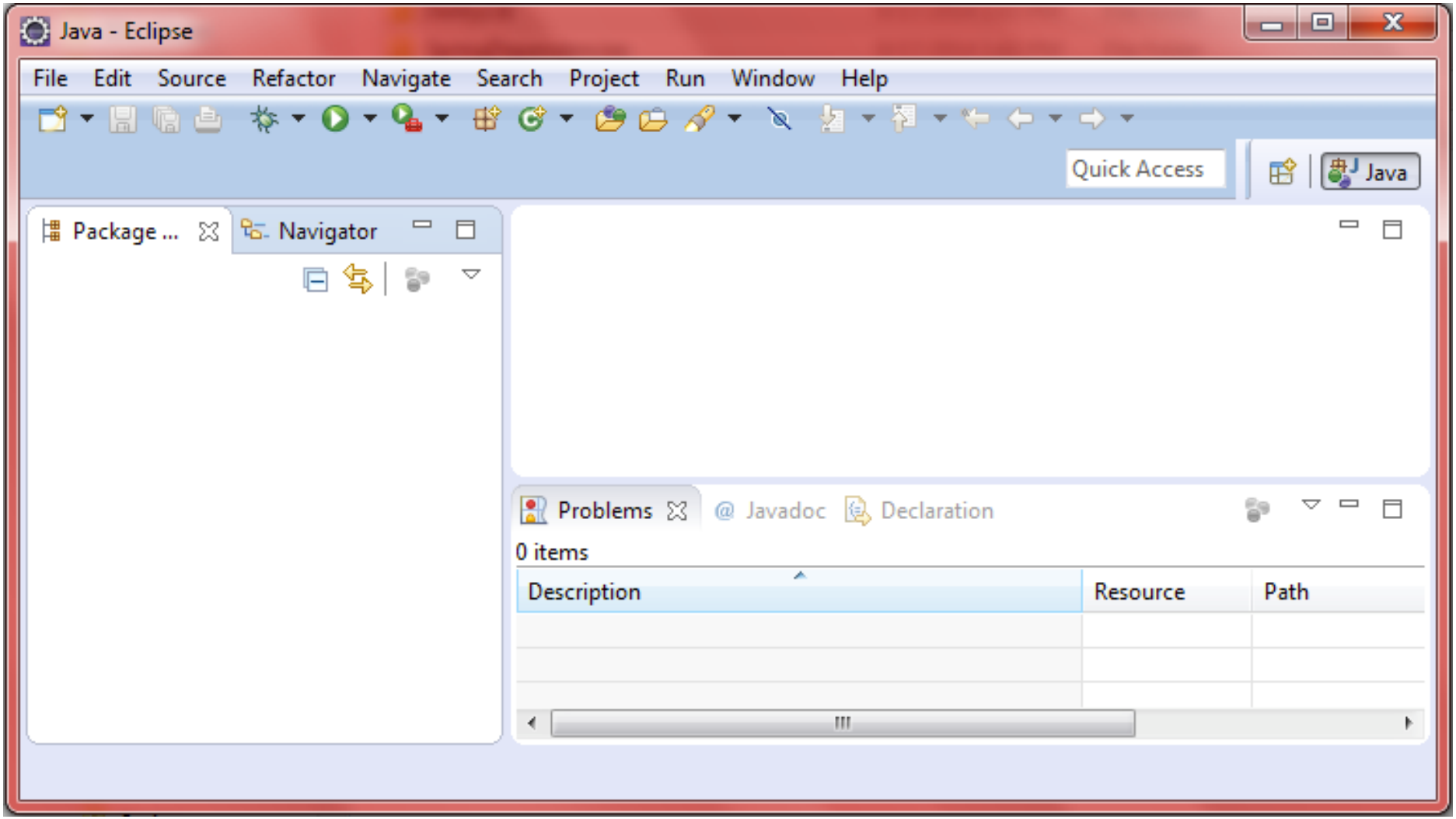




Java Perspective: Where's the Projects?

Lab

- Relax! The projects will reveal themselves as we proceed
  - Do not import anything



Create User Libraries

Lab

- Projects require Spring jars from the Spring distribution, plus external dependencies (all supplied in the lab setup)
  - We'll set up jars as an Eclipse user library for ease of use
  - Labs before Spring Boot depend on these libraries being set up
  - Later labs use maven for dependencies

Tasks to Perform

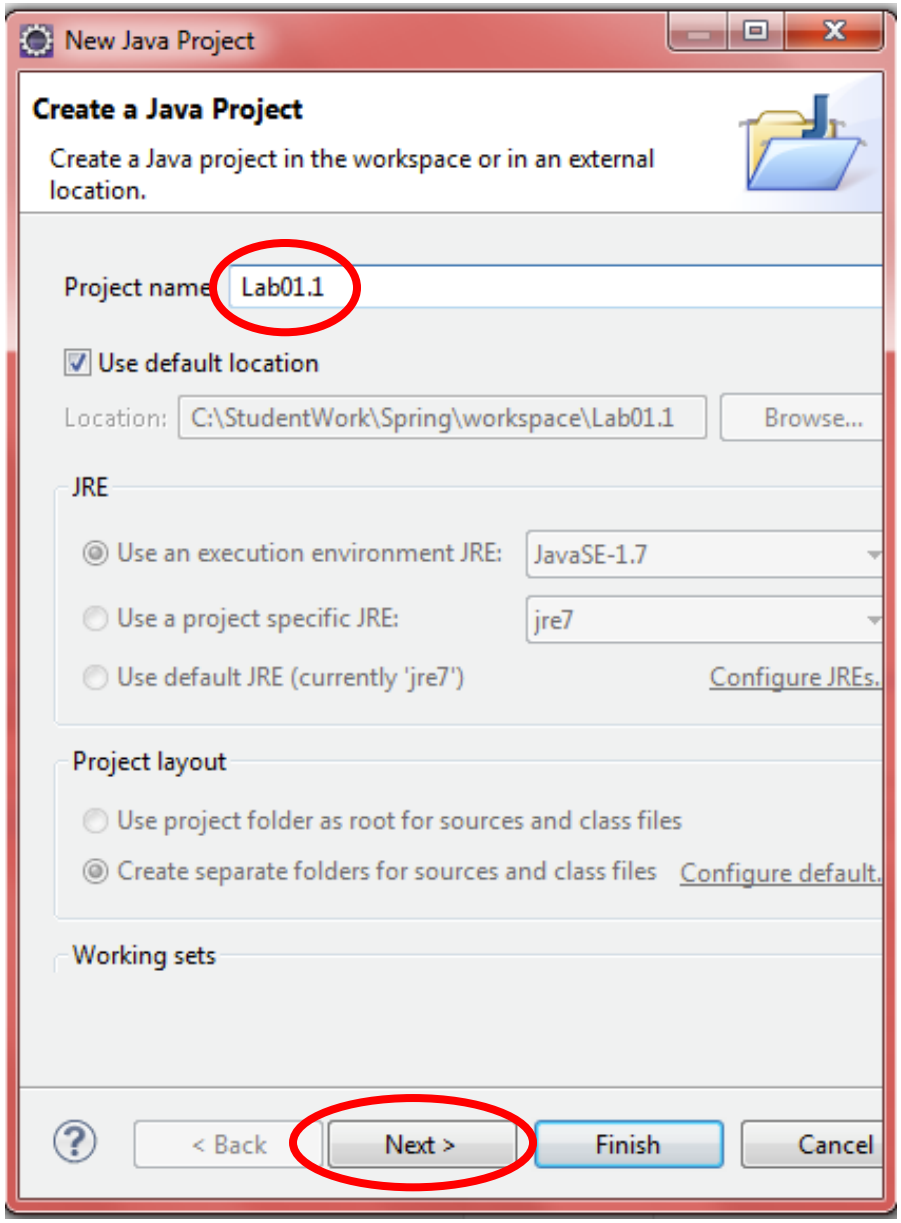
- Go to **Window | Preferences | Java | Build Path | User Libraries**
  - Click **New…** , in the dialog, call the library **Spring**, and press **OK**
  - Click the **Add External JARs…** button, browse to *StudentWork\Spring**SpringLibraries***
  - Select all the jars in that folder, Click Open
  - Select the Spring library again, click **Add External JARs…**, browse to *StudentWork\Spring**SpringDependencies***
  - Select all jars in that folder, Click **Open**
  - Click **OK** to finish creating the library

Create a Project for our Application

Lab

Tasks to Perform

- Create a **Java Project** (1)
  - Call the project **Lab01.1**
    - You MUST call it this to pick up starter files
  - Eclipse will then automatically set the project folder to *Lab01.1*
    - It already exists in the workspace
    - Contains Java project for Eclipse
  - Click **Next**
- This will bring you to the Java Settings dialog
  - It should show the src and test folders as source folders (2)
  - It will also let us add a library

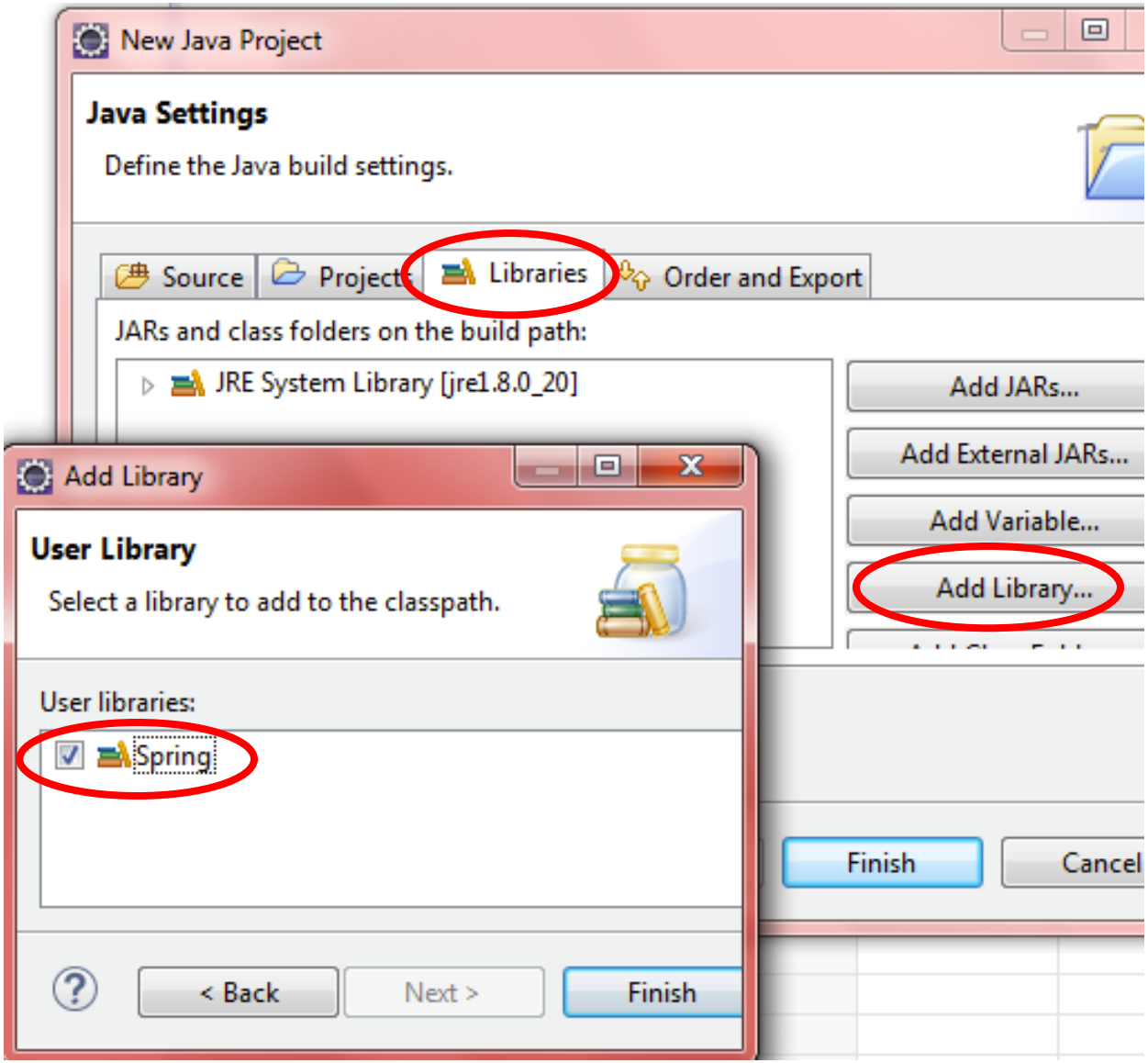


Add the Spring Library to the ClassPath

Lab

Tasks to Perform

- In **Java Settings** dialog, click the **Libraries** tab (1)
  - –Click **Add Library** button, and in the dialog that comes up, select **User Library** then click **Next**
  - –Check off **Spring**, then click **Finish** out of all dialogs

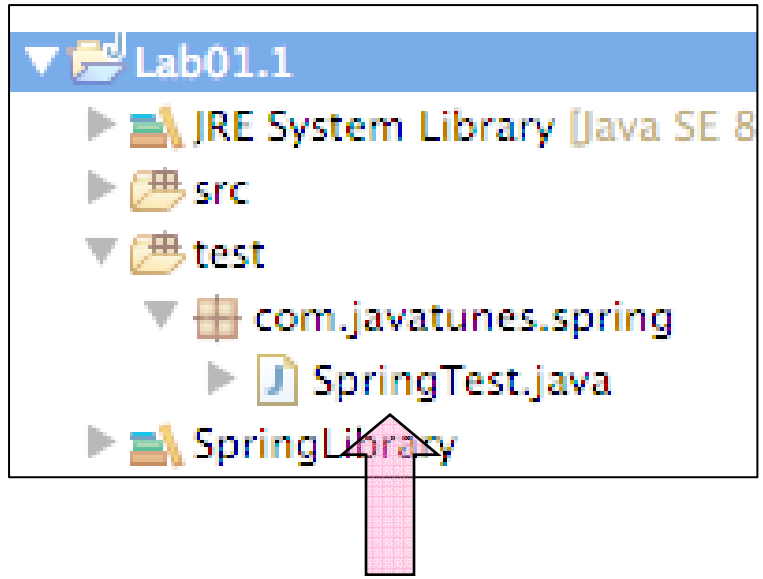


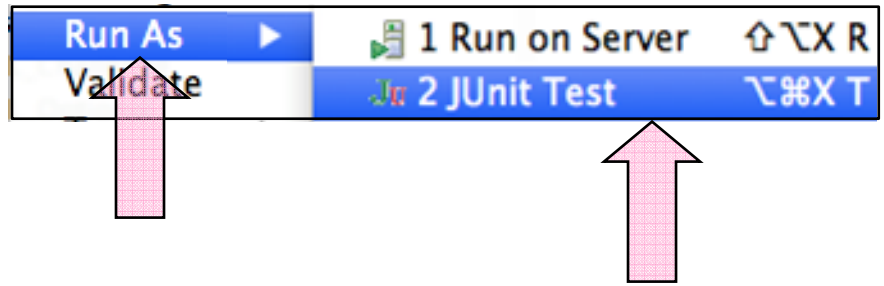
Running the Lab

Lab

Tasks to Perform

- We use Eclipse to run a test case in *SpringTest.java*
  - **No coding needed** in this class (1)
  - It just bootstraps Spring to test the environment
- After a clean build (error-free, not necessarily warning-free), do the following
  - In **Package Explorer** , right click on *SpringTest.java*
    - In `com.javatunes.spring` package under the test folder
  - Select: **Run As -> JUnit Test**
  - This automatically finds and runs *SpringTest*'s test method

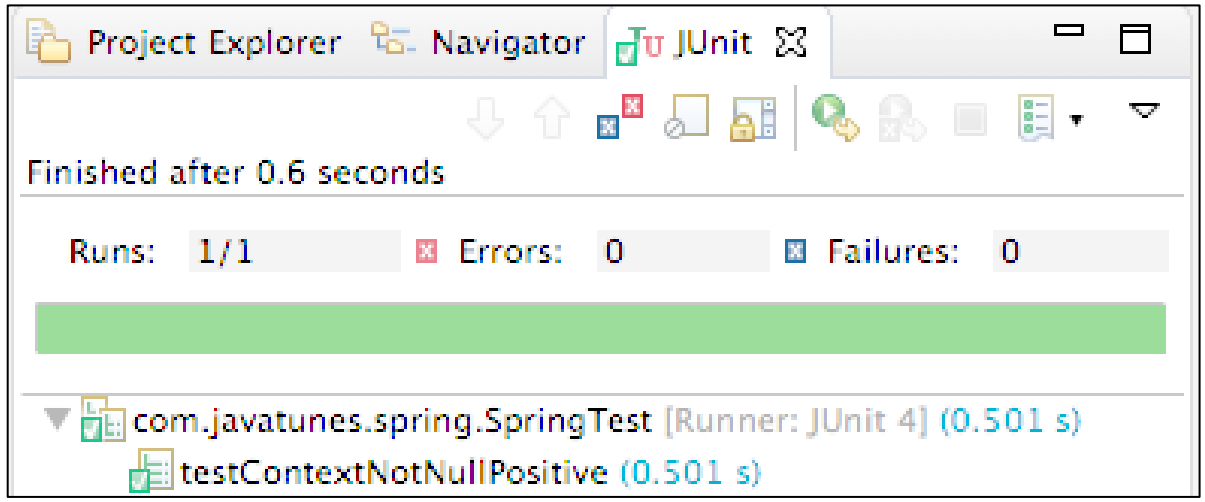




Program Output

Lab

- You should see JUnit output in a JUnit view, as shown below, and output in the Eclipse Console view as shown at bottom
  - These should all be error free (1)
- **Note**: You'll follow similar procedures **whenever you have a JUnit test to run** in the labs (2)



STOP

LEARNINGPATTERNS

Lab 1.2: Hello Spring World

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

Lab Synopsis

- **Overview**: In this lab, we will:
  - Become familiar with the different parts of basic Spring
  - Create and use a Spring context to access a bean instance
  - Write and run a simple Spring test

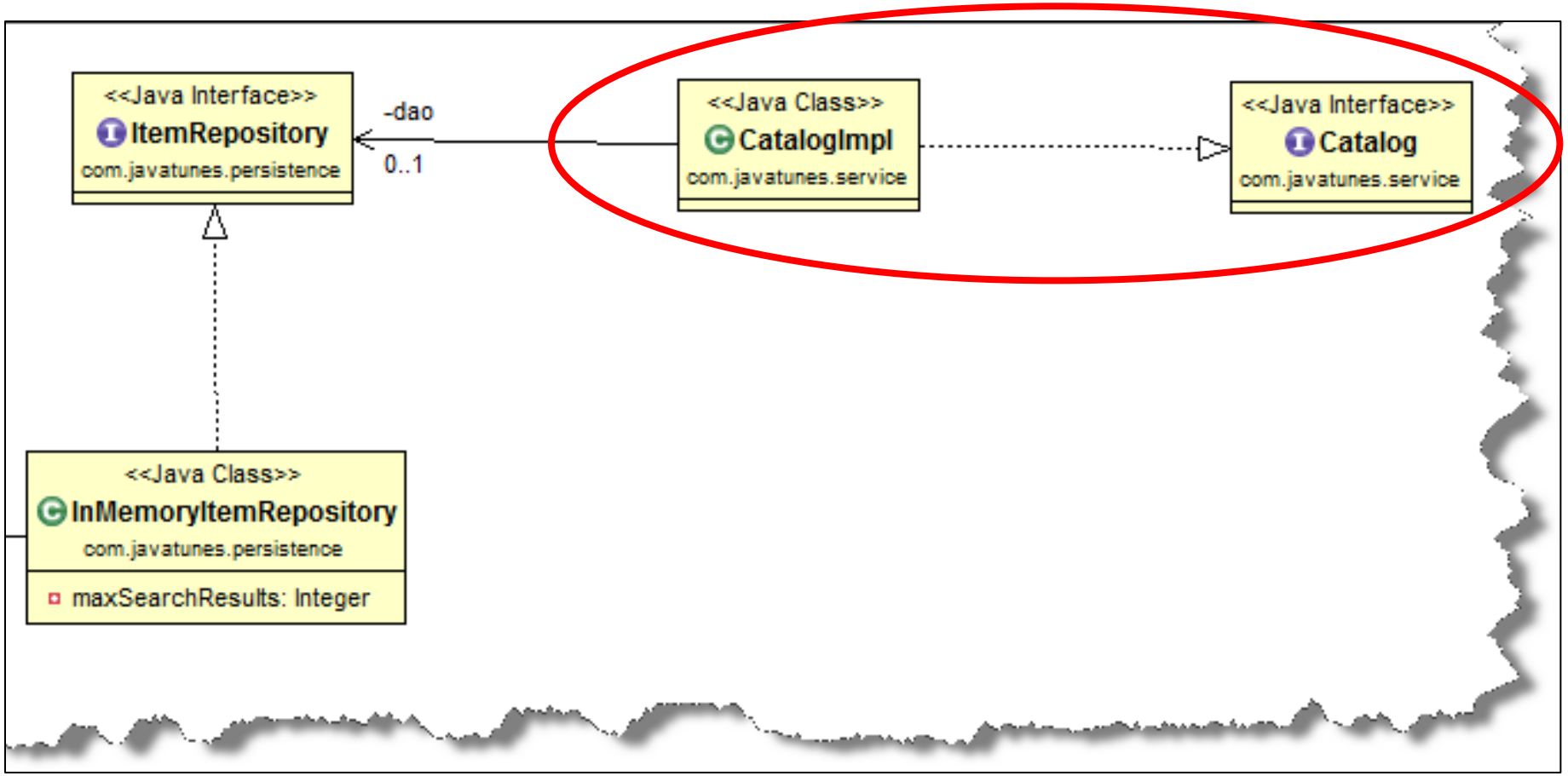
- **Builds on previous labs**: None

- **Approximate Time**: 20-30 minutes

Object Model

Lab

- **Object Model**: Our focus will be on the following types
  - We'll cover more types shortly
  - Note: Search methods in the catalog won't work yet



Lab Preparation

- The new lab folder where you will do all your work is:

  workspace\Lab01.2

Tasks to Perform

- **Close** all open files and projects

- **Create a new Java project** called Lab01.2 in the workspace
  - See Lab 1.1 instructions on how to do this if you need to
  - Remember to **add the Spring user library**
  - You can expect to see **compiler errors** until the lab is completed.

Writing and Configuring a Bean in Spring

Lab

Tasks to Perform

- **Open `CatalogImpl`** for editing
  - The class is in the `com.javatunes.service` package, under ***src***
  - Make sure **`CatalogImpl`** implements the **`Catalog`** interface
  - Remember - we code to interfaces to decouple from a specific implementation
  - **Save** your changes

- **Open *applicationContext.xml*** in the *src* folder for editing
  - Click the **Source tab** to edit
  - Finish up the declaration of the <bean> element by declaring:
    - An id of **`musicCatalog`**
    - A class of **`com.javatunes.service.CatalogImpl`**
  - **Save** your changes

Using a Bean in a Program

Lab

Tasks to Perform

- **Open** `com.javatunes.service.CatalogTest` (*test* folder) and:
  - In `testCatalogLookupPositive()`, modify the constructor for `ClassPathXmlApplicationContext`
    - Specify the `applicationContext.xml` file.
  - Next, look up the `musicCatalog` bean (by type) from the context (1)
    - `Catalog catalog = ctx…`
  - Make sure the catalog bean is not null (use `assertNotNull`)
  - Output the catalog bean (Just use `System.out.println()`)
  - Finally, call `close()` on the context and fix any compilation errors
- **Run** *CatalogTest* as a unit test and make sure the test passes
  - **Right click | Run As …| JUnit Test** as in previous labs
  - Your tests should pass, and you should see some console output
- You've successfully configured and used a bean with Spring
  - Congratulations !

Logging and Additional Things (Optional)

Lab

Tasks to Perform

- **Open** *log4j2.xml* in the *src folder*
  - This configures some of the logging that Spring will do
  - The root logger is configured at **`error`** level, spring at **`info`**
  - You can decrease Spring logging by configuring it at **`warn`**

```
<Logger name="org.springframework" level="warn"/>
```

  - **debug** level increases the logging - try some levels
- **[Optional]** Other things to try
  - Try looking up the bean by name, – what happens? What about if you look it up by the wrong name?
  - Try looking up a `CatalogImpl` instead of a `Catalog`
    - Does this work? Is it a good idea? (see notes)
  - Change your code back to your original solution before proceeding

LEARNINGPATTERNS

Lab 1.3: Spring Annotations

In this lab, we will use Spring annotations to configure beans

Lab Synopsis

- **Overview**: In this lab, we will become familiar with Spring annotations for declaring beans. We will:
  - Configure Spring to scan packages for annotated Spring beans
  - Annotate `CatalogImpl` to be a Spring bean
    - Using Spring annotations
  - `CatalogTest` does not need to change at all
    - It can simply look up the catalog as it did before
- **Builds on previous labs**: 1.2
  - Continue to work in your **Lab01.2** project
- **Approximate Time**: 20-30 minutes

Configuring an annotated Bean in Spring

Lab

Tasks to Perform

- Continue to work in the **Lab01.2** project
- **Open** `CatalogImpl`
  - Declare this as a Spring bean with `@Component`
    - Use the same bean id you used previously - `"musicItem"` (1)
    - Remember to import `@Component`
- **Open** *applicationContext.xml* (in the *src* folder)
  - Note the bean and context namespaces and schema locations at the top of this file
  - Comment out the bean declaration from the previous lab
    - Highlight the line, **Right click**, select **Source | Toggle Comment**
  - Add a line to configure scanning for annotated classes
    - The base package to start scanning from is `"com.javatunes"`

Check that Your Bean Exists

Lab

Tasks to Perform

- **Run CatalogTest** (in the ***test*** folder) and make sure it still passes
  - The output should be identical and all asserts should pass
- **[Optional] Uncomment** the bean declaration in *applicationContext.xml*
  - **Run** `SpringTest` again…did you see any errors?
  - Is this a good idea? Why or why not?
  - What happens if you name the bean in the XML something different, like "`musicCatalog2`"?
  - Do you get two instances of Catalog, or two references to one bean?
  - How could you confirm your conclusion in `SpringTest`?
  - Change your code back to your original solution before going on

LEARNINGPATTERNS

Lab 1.4: Dependency Injection

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

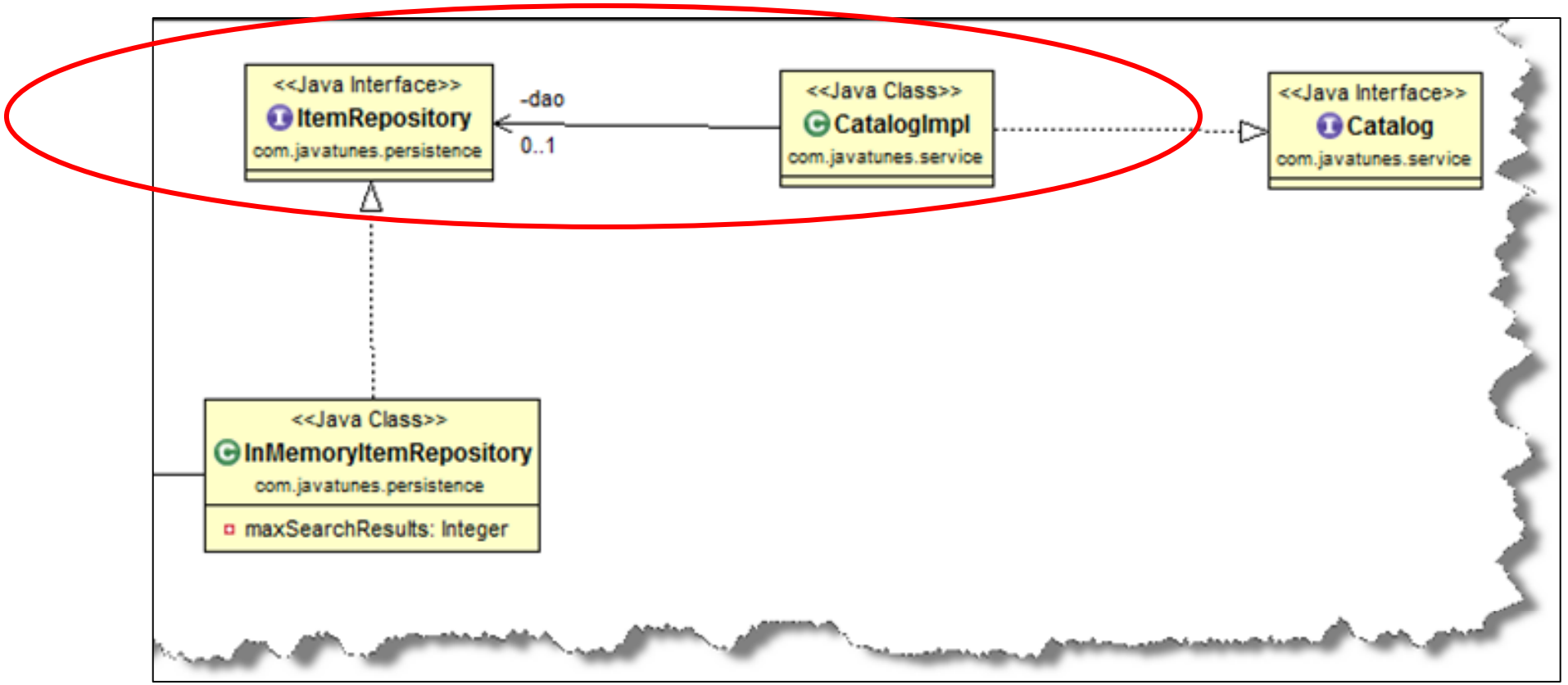
Lab Synopsis

- **Overview**: In this lab, we will become familiar with Spring DI and work with its capabilities. We will:
  - Configure Spring to inject dependencies based on the following:
    - `CatalogImpl` needs an `ItemRepository`
    - `InMemoryItemRepository` is a concrete implementation of `ItemRepository`
  - `CatalogTest` can look up the catalog as it did before
    - The catalog will now have complete functionality
    - And we can add other testing for it if we want
- **Builds on previous labs**: 1.3
  - Continue to work in your **Lab01.2** project
- **Approximate Time**: 20-30 minutes

Object Model

Lab

- **Object Model**: We'll focus on injecting an `ItemRepository` into the `CatalogImpl`
  - We don't cover `ItemRepository` details - it's given to you in full



Writing and Configuring a Bean in Spring

Lab

Tasks to Perform

- Continue to work in the **Lab01.2** project
- **Open** *InMemoryItemRepository class in com.javatunes.persistence* for editing
  - **Annotate this as a Spring bean**
  - It requires no injection or other configuration
- **Open** *CatalogImpl* in *com.javatunes.service*
  - Notice it has an `ItemRepository` property
    - A dependency on `ItemRepository`, which will need to be wired
  - **Add an annotation** to the property to "**autowire**" it
    - To inject it by type

Check if Your Bean Works Correctly

Lab

Tasks to Perform

- **Run CatalogTest** (in the ***test*** folder) and make sure it still passes
  - The output should also indicate that the catalog has a repository (You'll see this if you've output `Catalog.toString()` to the console)
- **Open** `CatalogTest` and find **`testCatalogPositive()`**
  - Annotate it as a test method, and add the following tests
  - Call the `size()` method on the catalog, check (with a JUnit assert) that it's not zero, then print out its value
    - It gives a result now - where did the data come from?
  - Call catalog's `findByKeyword("a")`
    - What is returned? Check (with a JUnit assert) that at least one item was found
    - Print out the returned data
- Run your test class again and check the results for errors

Break Your Program and Test It

Lab

Tasks to Perform

- In `CatalogImpl`, try commenting out the annotation for the dependency wiring (the injection)
- **Run your test again** - What happens? Why?
- Change your code back to your original solution before going on

LEARNINGPATTERNS

Lab 2.1: Java-based Configuration

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

Lab Synopsis

- **Overview**: In this lab, we will use Spring's Java-based configuration to configure our beans and dependencies
  - Providing Java-only metadata (no XML at all)
  - We'll create a separate `@Configuration` class for each tier
    - We have many options here !
  - We will import Java-based configurations into one root configuration
- **Builds on previous labs**: none
- **Approximate Time**: 25-45 minutes (depending on optional parts)

Lab Preparation

- The new root lab directory where you will do all your work is:

  workspace\Lab02.1

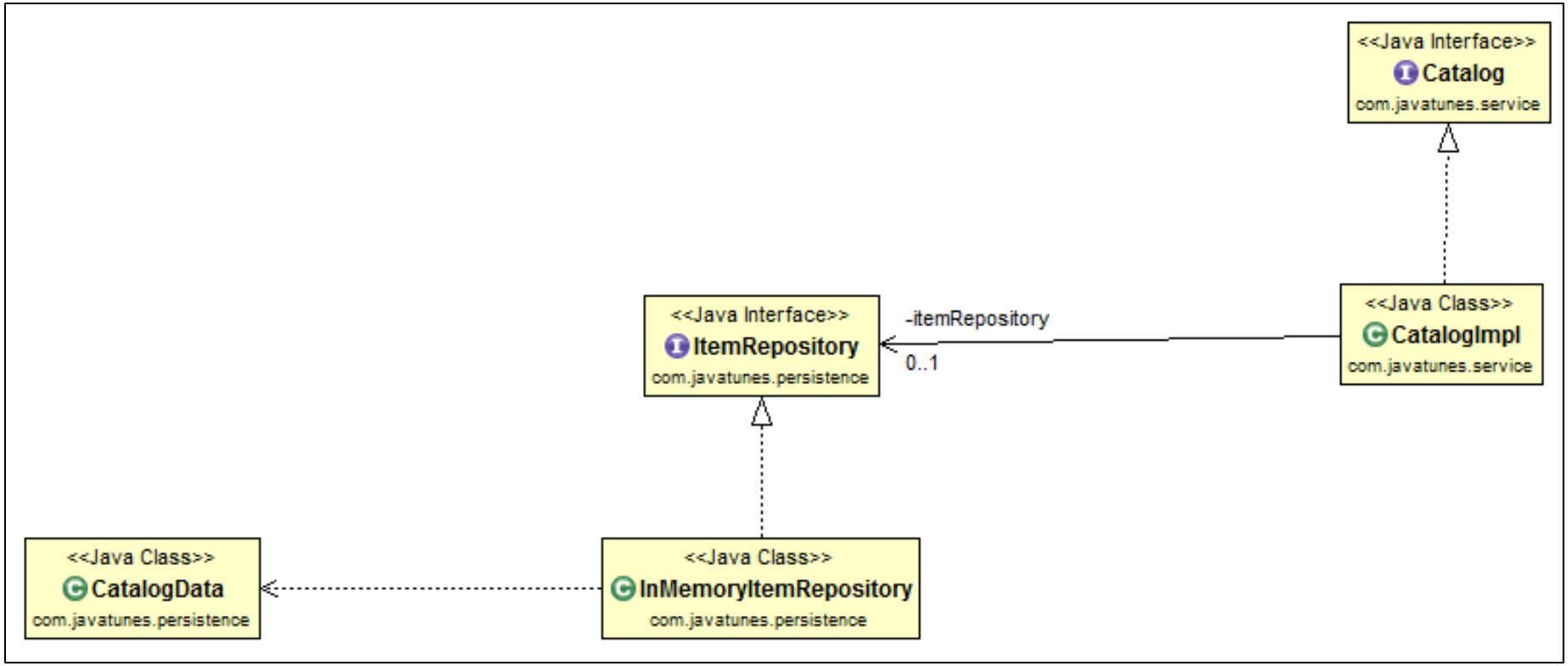
- **This lab contains no XML**
  - All configuration metadata will be written in Java - The safer XML!
  - Also, metadata will be external to our domain model

Tasks to Perform

- **Close** all open files and projects
- Create a new Java project called Lab02.1 in the workspace *
  - See Lab 1.1 instructions on how to do this if you need to
  - Remember to **add the Spring user libraries**
- **Don't worry** about errors (Red x's) and warnings
  - They'll soon disappear

Domain Model

Lab



Define Root Configuration Class

Lab

Tasks to Perform

- We supply the following configuration classes in this lab
  - `SpringConfig`: It imports the other configuration classes
    - It is the collector of the other classes that have bean defs in them
    - This allows us to bootstrap Spring with a single class!
  - `SpringRepositoryConfig`, and `SpringServicesConfig` contain bean definitions
  - Ignore `SpringDomainConfig` for now - it's used later

- **Open** *SpringConfig.java* for editing
  - Add the `@Configuration` annotation to the class
  - Add the `@ComponentScan` annotation with "`com.javatunes`" as the base packages
  - `@Import` the 2 Java config classes we'll use now (repository/service) (1)

Java-based Configuration

Tasks to Perform

- For the bean defs and wiring, look at the UML earlier in the lab
  - We have `ItemRepository` and `Catalog` implementations
    - And `CatalogImpl` needs an `ItemRepository`
  - We'll define/wire all of these using the `@Configuration` style (1)
    - We've removed the `@Component`/`@Autowired` annotations from our types for this lab
- **Open** *SpringRepositoryConfig.java* for editing
  - Add the **`@Configuration`** annotation to the class
- **Add an `@Bean` definition** that returns an **`ItemRepository`**
  - It should be named `itemRepository`
  - Return an instance or our `InMemoryItemRepository` class
- The item repository definition is the only `@Bean` in this class

Java-based Configuration

Tasks to Perform

- **Open *SpringServicesConfig.java*** for editing
  - Add `@Configuration` to the class
  - Note that we have a repository property of type `ItemRepository`
  - Use `@Autowired` to inject the repository

- **Add an `@Bean` definition** that returns a `Catalog`
  - Return an instance of our `CatalogImpl` class
  - Set its repository instance using the injected repository
    - Again - that is normal Java code using the (injected) repository variable in the class
  - This is the only `@Bean` in this configuration class

Finish Test Program and Run

Lab

Tasks to Perform

- Open `CatalogTest` for editing
  - You need to create the application context in both test methods
  - It's an `AnnotationConfigApplicationContext` now, since you're using `@Configuration` style
- **Run `CatalogTest`** as a JUnit Test
  - All tests should pass (they are the same as the last lab)
  - You should see items printed out due to the lookup in the test
  - Up to `InMemoryItemRepository.maxSearchResults`, which defaults to 30 items

[Optional] Java Code in Config Class

Lab

Tasks to Perform

- [Optionally] We can put other Java code in a config class
  - All Java's power is available, as we'll show here
  - The metadata Java code may not be executed "directly" but it is indirectly executed, and we can use it !

- **Open** *SpringDomainConfig.java* for editing
  - Add the `@Configuration` annotation to the class
  - Add @Bean to the existing `maxSearchResults()` method
    - Note how it contains simple code returning a **randomly generated** Integer
    - It could be a more complex calculation, doing whatever we wanted

- **Open** *SpringConfig.java* for editing
  - Add `SpringDomainConfig` to the list of imported configuration classes (1)

[Optional] Inject the Integer Value

Tasks to Perform

- **Open** *SpringRepositoryConfig.java* for editing:
  - We'll set a field in the repository using an injected value
  - We could do this differently (1)
- **Add** a `private Integer` field called **`maxSearchResults`**
  - Annotate it with **@Autowired** to initialize it
    - This will inject the value configured in `SpringDomainConfig` (2)
- Do the following in the `@Bean` def returning an `ItemRepository`
  - After creating the `InMemoryItemRepository`, set its `maxSearchResults` property
    - Use the `maxSearchResults` field you just injected into the config class
  - Return the repository instance as before
- **Run** `CatalogTest` as a JUnit Test again
  - You should see a random number of Items returned each time you run the test

Summary

- Java based configurations provide you with the best-of-breed style of specifying your application requirements.
  - It's Java code so IDE support is mature and solid
  - You effectively write the code you have always written to satisfy dependencies
    - But not in your application code!
  - You can have logic that determines what happens in real-time
  - And refactoring and cross-referencing is clean and simple
- This has a good chance of become the preferred technique
  - With XML being used narrowly, and annotations used to eliminate ambiguities (covered next)
  - The Spring project now recommends this
- It's evolution !

LEARNINGPATTERNS

Lab 2.2: Bean Lifecycle

In this lab, we will work with the bean lifecycle

Lab Synopsis

- **Overview**: In this lab, we will work with the bean lifecycle
  - Explore and configure different bean scopes
  - Work with some of the bean lifecycle callbacks and the interfaces Spring provides to work with them
- **Builds on previous labs**: Lab 2.1
  - Continue working in your Lab02.1 project
- **Approximate Time**: 20-30 minutes

Prototype Scope

Lab

Tasks to Perform

- **Open `SpringServicesConfig`** for editing
  - Change the scope of the catalog bean to prototype
  - **Open `CatalogTest`** and in `testCatalogLookupPositive()` get a second catalog

```
Catalog catalog2 = ctx.getBean(Catalog.class);
```

  - Add an `assertTrue` that asserts (`catalog != catalog2`)
    - We've imported all the assert methods already (via `import static`)
- **Run the tests**, make sure that they pass
- **Add a** creation lifecycle callback
  - **Open `CatalogImpl`** for editing
  - Add a method **`public void init() { … }`**
  - Annotate it with **`@PostConstruct`**
  - In the method, just print out something using `System.out.println()`

Bean Lifecycle

Lab

Tasks to Perform

- **Run** `CatalogTest` again
  - Go to the console view in Eclipse - do you see any output?
  - How many catalogs were created?

- In *SpringServicesConfig.java*, **remove the prototype scoping** on the catalog
  - Run the tests again - this time you should have one failure on the assert of `catalog != catalog2`
  - Why? What is default scope of a Spring managed bean?
  - Remove the assert for this in `CatalogTest`, and run it again (1)
  - Does it pass? How many catalogs are created?
  - That's it - you've used the lifecycle methods to see how scoping works !

LEARNINGPATTERNS

Lab 2.3: Profiles

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

Lab Synopsis

- **Overview**: In this lab, we will create profiles to customize our bean definitions
  - We'll create two profiles that have different configurations, then test both of them
  - We'll also (optionally) try out inheritance in `@Configuration` classes
- **Builds on previous labs**: none
- **Approximate Time**: 20-45 min. (Depending on optional parts)

Lab Preparation

- The new root lab directory where you will do all your work is:
  workspace\Lab02.3

Tasks to Perform

- **Close** all open files and projects
- **Create a new Java project** called Lab02.3 in the workspace *
  - See Lab 1.1 instructions on how to do this if you need to
  - Remember to **add the Spring user library**
- Note: This lab combines `@Bean` style bean definition and `@Autowired` style DI with @Configuration style configuration

Set up Profiles

Lab

Tasks to Perform

- **Open** `SpringDevRepositoryConfig` in `com.javatunes.config` for editing
  - Annotate it to be profile "**`dev`**" (1)
  - Finish the **`itemRepository()`** method
    - Create an instance of `InMemoryItemRepository`
    - Set the **`maxSearchResults`** with the value from the autowired property above
    - Return the instance
- Repeat similar steps in `SpringProdRepositoryConfig`
  - Annotate it to be profile "**`prod`**"
  - Create a `ProductionRepository` instance in **`itemRepository()`**
  - Return the instance

Activate and Test Profiles

Tasks to Perform

- **Open `CatalogTest`** in the *test* folder for editing
  - Review a test method to see how it calls **`setupContext()`**
  - Find the **`setupContext()`** method near the top - it sets up the profile and reads the config class for all the test methods
  - **Set the active profile** to the passed in profile name
  - **Register the configuration class** that was passed in
  - **Refresh the context** and save your changes

  - Review the rest of the test class - it has test methods for both the dev and production profiles
- **Run** *`CatalogTest`* as a JUnit Test
  - You should see output from both profiles (1)
- Congratulations! You have switched between two different profiles.

[Optional] Active Profile via Property

Lab

Tasks to Perform

- **[Optional]** Use a different method of setting the active profile
  - First comment out the setting of the active profile in the dev test case (
  - Run the test and see how it works
  - It should fail, since the catalog bean won't be defined if none of the profiles is active
- Set the profile to "**dev**" using the `spring.profiles.active` property
  - You can set this as a VM property in the Run Configuration (1)
- **Run** the test case again
  - You should see output indicating the appropriate profile was selected

[Optional-2] @Configuration Inheritance

Tasks to Perform

- Review the two repository config classes
  - Do you see any commonality that you can factor out into a base class
    - Review the inheritance example in the manual slides for ideas
    - It's OK if it's very simple - we purposely keep the types simple
- Create an `@Configuration` class - `SpringBaseRepositoryConfig`
  - Put the common code from the two configuration classes in it
  - Add any new needed code to this base class
- Modify the other two config classes to extend the base class
  - Remove the common code from both types
  - Add to or modify these two config classes to work correctly
- **Run** the test case again
  - All tests should still pass

LEARNINGPATTERNS

Lab 3.1: Hello Boot World

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

Lab Synopsis

- **Overview**: In this lab, we will become familiar with Spring Boot and run a simple application that uses it
  - We'll provide all of the code and configuration
  - You will just run it, and examine how Spring Boot works
  - You will use maven and Eclipse to work with it
  - You'll also become familiar with the maven-based project structure
- **Builds on previous labs**: None
- **Approximate Time**: 20-30 minutes

Import maven Project into Eclipse

- Eclipse understands maven projects fairly well
  - The starter files (e.g. *pom.xml*, and Java source) are supplied by us
  - When importing, Eclipse examines *pom.xml*, and sets up the classpath
  - It understands maven's project structure, and sets up the Eclipse project
  - It will compile the source as normal when importing (or changing source)
    - No need to use maven for a compile or program execution- Eclipse does it

Tasks to Perform

- **Import** the Lab03.1 project into Eclipse as follows:
  - **File | Import … | Maven | Existing Maven Projects**
  - Click **Next**, Browse to the **workspace\Lab03.1** folder, click **Finish**
- **You may have errors** in the project
  - You **may** (or **may not**) be able to reach the maven repositories
  - It depends on your environment setup (e.g. firewalls)
  - We'll check that, and resolve any problems - see next slide

Check maven Repository Access

Lab

Tasks to Perform

- Check the console window for error messages
  - **If you see an error** like that below - starting with "Project build error: **Non-resolvable parent POM** …", you can't reach the maven repository
    - Likely due to firewall/security restrictions
- If you see this error, follow the instructions on the next slide
  - See notes on this page if you can't get maven to work any way at all (1)
- If you **do NOT see** such an error, then skip the next slide
  - It indicates you can reach the maven repositories

Setup maven Repository Access

Lab

Tasks to Perform

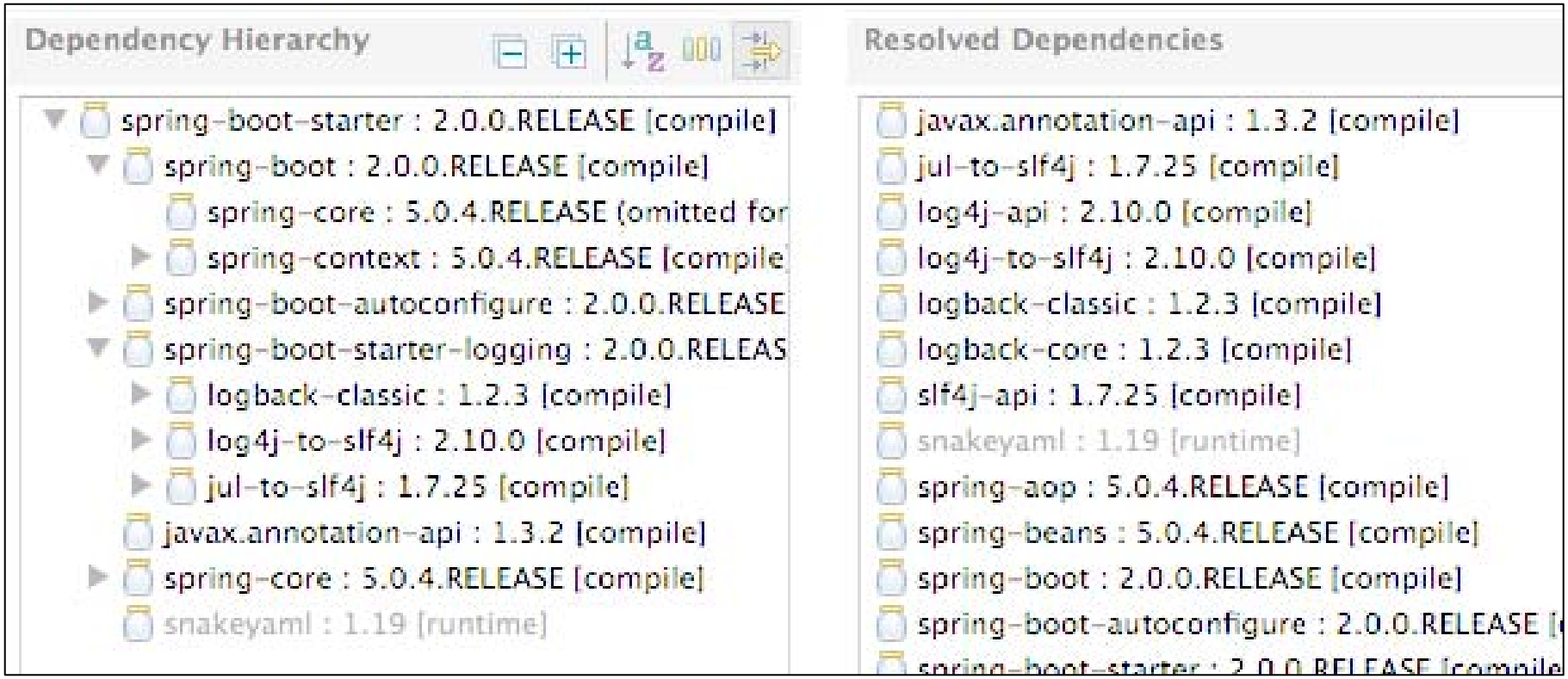
- Only perform these steps **if you can't access the maven repository**
- To resolve the error, you will replace your maven repository
  - With one that we supply, that is pre-populated with all needed libraries
- Go to your home folder (1), and find a folder named **.m2** ("dot"m2)
  - If it exists, rename that folder to .m2-ORIG (2)
  - If it doesn't exist, that's no problem, just continue below (but read the note)
    - **Copy** the ***.m2*** folder in *StudentWork\Spring**mavenRepository***
  - **Paste** it to your **home folder**
  - That's it - you've got a local maven repo with all the jars you need
- Update the Lab03.1 project - **Right Click | maven | Update Project**
  - The error should now go away - your maven repo is set up
- Remember to **restore your original** .m2 folder after the course
  - Rename the ***.m2*** lab repo to ***.m2-Spring***, and rename ***.m2-ORIG*** to ***.m2***

Review the Project POM

Lab

Tasks to Perform

- Within Eclipse, open *pom.xml* for review
  - First view the straight source (*pom.xml* tab)
  - The **Dependency Hierarchy** tab shows direct/indirect dependencies



Run the Application

Lab

Tasks to Perform

- Right click on **HelloBootWorld.java** (in Eclipse Package Explorer)
  - Select **Run As | Java Application**
  - It should run cleanly, and produce output like that below
  - Note output from `main()`, and from `run()` (`CommandLineRunner`)
  - Open `HelloBootWorld` for review to see how these are produced

```
HelloBootWorld.main() called

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

2018-03-19 19:36:25.160  INFO 90757 --- [           main] com.example.HelloBootWorld
2018-03-19 19:36:25.164  INFO 90757 --- [           main] com.example.HelloBootWorld
2018-03-19 19:36:25.842  INFO 90757 --- [           main] com.example.HelloBootWorld
ARunner.run
```

Review Debugging Output

Lab

Tasks to Perform

- Spring Boot has copious debugging built in
  - Open *src/main/resources/**application.properties***, set boot logging to debug

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

  - **Rerun** - this prints out a lot of info - e.g. the auto-configuration (see below)

```
Positive matches:
-----------------

   GenericCacheConfiguration
      - Automatic cache type (CacheCondition)

   JmxAutoConfiguration
      - @ConditionalOnClass classes found: org.springframework.jmx.expo
      - matched (OnPropertyCondition)
```

```
Negative matches:
-----------------

   ActiveMQAutoConfiguration
      - required @ConditionalOnClass classes not found: javax.jms.Co

   AopAutoConfiguration
      - required @ConditionalOnClass classes not found: org.aspectj.
```

Create Executable Jar

Lab

Tasks to Perform

- We will package the project with maven from within Eclipse (1)
  - Right click on the project, select **Run As | Maven build ...**
    - Name the Run configuration **`Lab03.1-package`**
    - Enter a goal of **`package`**
    - Select **Apply,** then **Run** - This runs the **mvn package** command
    - See notes for possible errors
  - Creates a jar (the default maven packaging) in the ***target*** folder
  - We've included the spring boot maven plugin in the POM
    - Review it in *pom.xml* - it has an artifactId of **`spring-boot-maven-plugin`**
    - It repackages the jar as an executable jar with all needed dependencies
    - The original (*.jar.original* extension) and repackaged jar (*.jar*) are both produced
  - Go to the Navigator view (2), **right click** on **Lab03.1**, select **Refresh**
  - You should see the jars that were created (in the ***target*** folder)

Note on maven Repositories

- We've occasionally seen odd errors in maven projects with Eclipse
  - Either compile or runtime
  - Hard to track, and hard to figure out where they come from
- Sometimes simply updating the project solves this
  - **Right Click | maven | Update project…**
- Sometimes there is something odd in your repository
  - So you may need to rename the existing repo to something else
    - e.g. .m2 to .m2.ORIG
  - Then update the project
  - This only works if you have access to the maven repo
- Keep this in mind if you get odd errors occurring

LEARNINGPATTERNS

[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

Lab Synopsis

- **Overview**: In this (optional) lab, we will work with a simple JUnit test case, including doing the following:
  - Set up an IDE project to use JUnit
  - Test a simple `Calculator` class using some basic assertions
- **Builds on previous labs**: None
  - The new root lab directory is ***workspace\Lab04.1***
- **Approximate Time**: 15-20 minutes

Lab Preparation - Bring in Project

Tasks to Perform

- **Close** all open files and projects

- **Import an existing Maven project** called Lab04.1 into the workspace
  - **File | Import … | Maven | Existing Maven Projects**
  - Click **Next**, Browse to the **workspace\Lab04.1** folder, click **Finish**
  - There is no need to add Java libraries to this project
  - **Maven** will pull in all dependencies needed via *pom.xml*

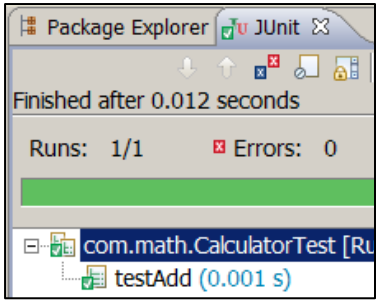
- Open *pom.xml* for review
  - We bring in the JUnit dependency, and that's about it
  - We don't need Spring or Spring Boot - but maven is useful even in this simple situation
  - It pulls in a number of dependencies that JUnit has

Review Lab Classes and Run Tests

Lab

Tasks to Perform

- Note the structure of the project and the names of the classes
  - Parallel *src* and *test* directories
  - Test classes and business classes in same package
- Review the `Calculator` class – yes, it's dumb-simple BUT:
  - Understand its API and what to **expect** from its methods
    - This forms the basis for your test cases
  - This simple class gives us a good example to start with
- Review the `CalculatorTest` test class
  - Run its test cases (methods) via right-click → **Run As → JUnit Test**
  - Eclipse JUnit view appears, showing a green bar



Write/Run More Tests

Lab

Tasks to Perform

- Write additional test methods for the `Calculator` class, following the example `testAddPositive()` method
  - `testDividePositive()`: Use 5/2 for your test
    - Use the 3-argument **`assertEquals(expected, actual, delta)`** method (because of the `double` return type) – for example:

```
assertEquals(2.5, calc.divide(5, 2), .001);
```

  - `testIsEvenPositive()`
    - Use **`assertTrue()`**, passing in a should-be-true condition – for example:

```
assertTrue(calc.isEven(10));
```

- **NOTE**: tests run in **isolation**, and in **no guaranteed order** (1)
  - You need to create a new `Calculator` object in each test

- Run the tests as before: right-click → **Run As → JUnit Test**

What Gives

Lab

Tasks to Perform

- You should have seen a failure in one of the tests (divide)
  - Look at `Calculator.divide()`- the arguments are int
  - int / int result is an int - so what happens if you have a remainder
    - It gets truncated
  - Change the implementation to the below

```
return 1.0 * a / b; // Convert to double, then divide
```

- Run the tests as before: right-click → **Run As → JUnit Test**
  - They should all pass
  - You can see, that even in a small class, there is room for error
  - That's why we test !
- **[Optional]** Add a negative test and run it
  - Note: You can run a single test in a test class also
    - Expand the test class in Package Explorer, **right click** on a single method, select **Run As | JUnit Test**

LEARNINGPATTERNS

Lab 4.2: Spring Testing

In this lab, we will work with the features of Spring Testing

Lab Synopsis

- **Overview**: In this lab, we will work with Spring Testing
  - We'll review the POM for it (Spring Boot-based)
  - We'll integrate our test class with a Spring context
- **Builds on previous labs**: None
  - The new root lab directory is ***workspace\Lab04.2***
- **Approximate Time**: 20-30 minutes

Lab Preparation - Bring in Project

Lab

Tasks to Perform

- **Close** all open files and projects

- **Import an existing Maven project** called Lab04.2 into the workspace
  - **File | Import … | Maven | Existing Maven Projects**
  - Click **Next**, Browse to the **workspace\Lab04.2** folder, click **Finish**
  - There is no need to add Java libraries to this project
  - **Maven** will pull in all dependencies needed via *pom.xml*

- Open *pom.xml* for review
  - We bring in the **`spring-boot-starter-test`**, and that's about it !
  - Look at the Dependency Hierarchy tab to see what is brought in
    - Quite a bit
    - Spring Boot makes life easy

Review and Complete Test Class

Lab

Tasks to Perform

- Briefly review the types under the *src* folder
  - They are the JavaTunes types we've seen earlier
- Open **`CatalogTest`** (`com.javatunes.service` under *test* folder)
  - Annotate the class to set the JUnit runner to Spring's custom runner
  - Annotate the class to read the config class `SpringConfig.class`
    - Use the annotation that **initializes Spring Boot**
    - And note how we inject a `Catalog` into the test class
  - We used a JUnit annotation to set the test method execution order (1)
  - It runs the methods in alphabetical order (`test1…`, `test2…` , `test3…`)
- Run the **`CatalogTest`** test methods
  - **Right-click | Run As | JUnit Test**
  - Hmm - some failures again - let's see what's up

Fix Test Class

Lab

Tasks to Perform

- What's the problem?
  - `test3_testSizePositive()` tests the number of items in the repo
    - We know what that should be in this repository
  - However, `test2_testDeletePositive()` deletes an item
    - And it runs before test3 (because we've set it up this way just for this)
- **Annotate** `test2_testDeletePositive()` to indicate that the **context should be recreated** after this method runs
  - See the manual slides for the correct annotation
- Run the **`CatalogTest`** test methods again
  - **Right-click | Run As | JUnit Test** - They should run without failure now
- Spring Boot and Spring Test make it very easy to do testing
  - Create your test class to use the capabilities and you're set
  - You can do everything the rest of your Spring program can

LEARNINGPATTERNS

[Optional] Lab 5.1: Integrating Spring and JPA

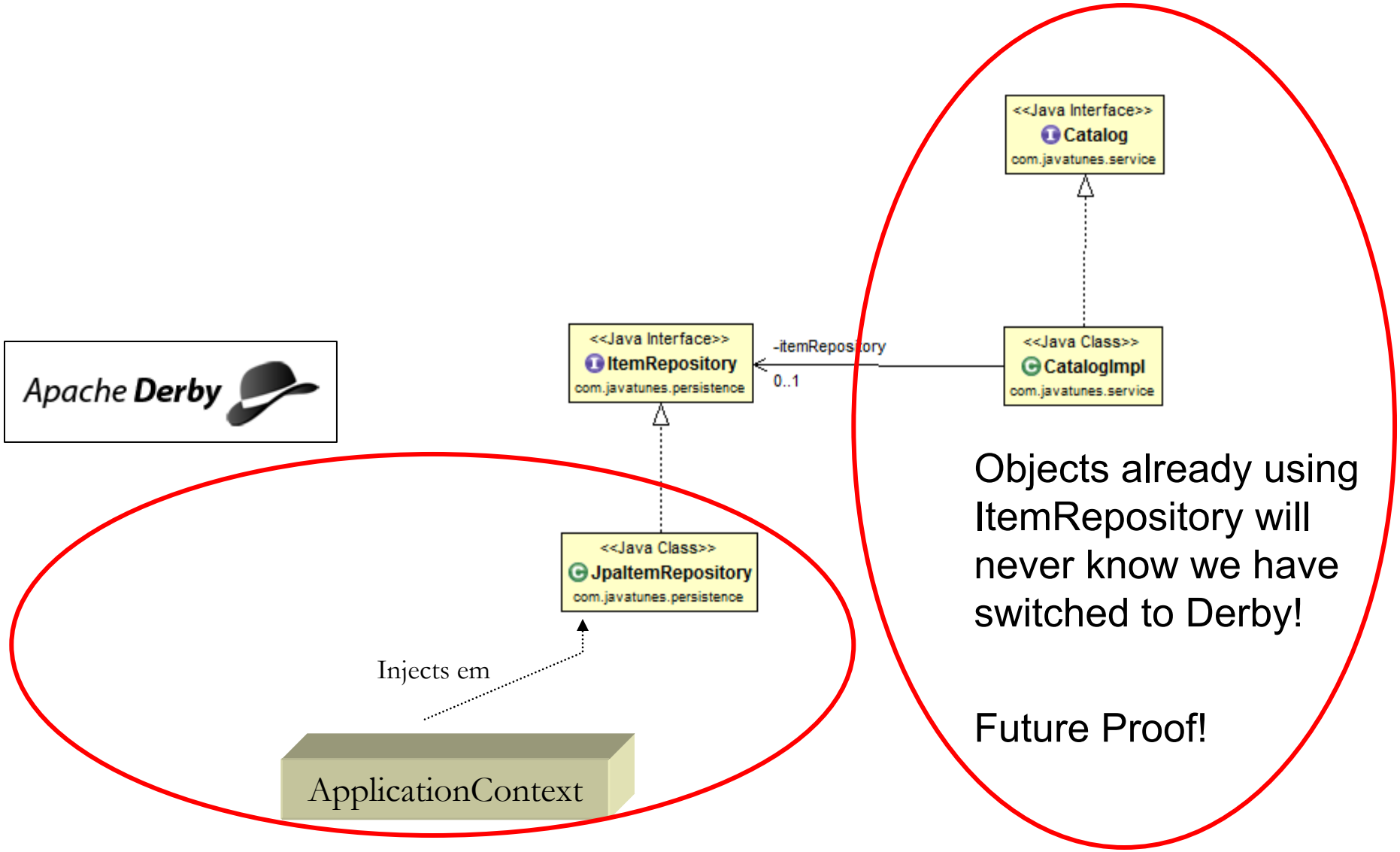
In this lab, we will work with the Spring/JPA integration

Lab Synopsis

- **Overview**: In this (optional) lab, we will work with the Spring/JPA integration, including
  - Setting up a `LocalContainerEntityManagerFactoryBean`
  - Finishing a JPA-based Repository class that injects an EM
- **Builds on previous labs**: None
  - The new root lab directory is ***workspace\Lab05.1***
- **Approximate Time**: 35-45 minutes
- **Note**: We will be using the Apache Derby database

Domain Model

Lab



The Moving Parts

Lab

- You will focus on the following classes in this lab
  - `MusicItem` : Our entity class - now persisted to the DB
  - `ItemRepository` : API to define data access
    - `JpaItemRepository` : Concrete implementation that uses JPA
  - `Catalog` : Client API for services provided
    - `CatalogImpl` : Concrete implementation of `Catalog`

- You will also work with the following Spring configuration
  - **Datasource**: Already configured to connect to our Derby database
  - **JpaVendorAdapter**: Will configure for Hibernate
  - **EntityManagerFactory**: We'll inject our datasource and vendor adapter

The Configuration

- We use `@Configuration` metadata in this lab
- The database configuration is done in `SpringRepositoryConfig`
  - In *com.javatunes.config*
  - Configuration strings are externalized in a properties file
    - Makes it easy to change configuration without recompiling
    - e.g. database, driver class, or URL
- Bean definitions and wiring are done using `@Configuration`
  - In `SpringRepositoryConfig` and `SpringServicesConfig`
  - Both are imported by `SpringConfig`
- This lab is structured as a Spring Boot/JPA/Maven project
  - With a *pom.xml* for maven (the only XML used)

Lab Preparation - Bring in Project

Lab

Tasks to Perform

- **Close** all open files and projects

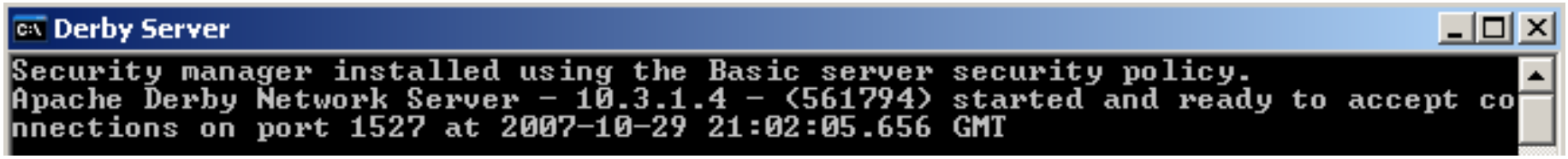
- **Import an existing Maven project** called Lab05.1 into the workspace
  - **File | Import … | Maven | Existing Maven Projects**
  - Click **Next**, Browse to the **workspace\Lab05.1** folder, click **Finish**
  - There is no need to add Java libraries to this project
  - **Maven** will pull in all dependencies needed via *pom.xml*

Setup/Create the Derby Database

Lab

Tasks to Perform

- **Open** the *StudentWork\Spring\Derby* folder (part of the lab setup)
  - See notes for Unix variants (1)
- **Start** the Derby database server:
  - Execute **`dbStart.cmd`** (you can double-click on it)
  - This starts a standalone Derby server that can accept network connections to the database
  - You should see output like that below in the command window



- **Create the database**:
  - Execute **`dbCreate.cmd`** (you can double-click on it)
  - This creates the **`JavaTunesDB`** database which we'll use in the labs

Finish Entity/Database Configuration

Lab

Tasks to Perform

- **Open `MusicItem`** for review (there is **nothing to do** here)
  - In `com.javatunes.domain` in the *src* folder
  - Note how the table name is specified (since it doesn't match the class)
  - Note the id and generation strategy
  - Note that the names of most persisted fields match the column names of the table, and therefore need no annotations (1)
  - Note that **`MusicCategory`** is Enumerated, and persisted as a string
  - Note that the **`DateTimeFormatter`** is transient
    - It it won't be persisted (used internally for formatting only)

- **Open** *jdbc.properties* in the *src* folder for review/editing
  - Look for the **TODO**s to complete the following:

    url: `jdbc:derby://localhost:1527/JavaTunesDB`
    persistenceUnitName: `javatunes`

Complete Repository and Configuration

Lab

Tasks to Perform

- **Open `SpringRepositoryConfig`** (`com.javatunes.config`)
  - Configure the reading of the *jdbc.properties* file (by a property source)
  - Configure injection of the environment
  - Configure lookup of any needed **properties** from *jdbc.properties*
    - Search for **`"TODO"`**, including the quotes, to find them
  - Add an `@Bean` declaration for the `itemRepository()` bean (1)
    - **Return type**: `ItemRepository`, and **actual object**: `JpaItemRepository`

- **Open `JpaItemRepository`** for editing
  - This is our JPA `ItemRepository` implementation
  - Inject the **`EntityManager`** (what annotation do you use?)
  - **Review** `get(Long id)` - It returns what the entity manager finds
  - Note: Most of the methods are not finished - that's OK
    - The lab focuses on the Spring configuration, not working with JPA

Finish Other Configuration

Lab

Tasks to Perform

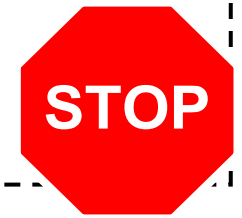
- **Open** *SpringServicesConfig.java* for editing
  - Inject into the `repository` variable
  - Complete the **`catalog()`** method, and declare it as a bean (1)
    - **Return type**: `Catalog`, **Actual object**: `CatalogImpl`
    - Remember to set the repository for the `CatalogImpl`
- **Open** *SpringConfig.java* for review (There is **nothing to do** here)
  - Note how it imports the `@Configuration` metadata
- **Open** *CatalogTest.java* for review (*src/test/java*)
  - Note the **`@SpringBootTest`** to bring in Spring Boot capabilities, and read our configuration
  - It injects the catalog bean, then uses `findById()` to get a `MusicItem`
    - Done by JPA, but this is transparent to the client
  - It then prints the returned `MusicItem` to the console

Test Program

Lab

Tasks to Perform

- **Run** `CatalogTest` as a JUnit test and view the output
  - It should successfully find the item by id and pass all tests
  - You should also see logging information from both the Spring and JPA runtimes (we've set the log level to info - see notes)
  - That's it – you've successfully configured and used a JPA-based Repository that uses an injected EM



LEARNINGPATTERNS

Lab 5.2: Spring Data

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

Lab Synopsis

- **Overview**: In this short lab, we will work with Spring Data
  - We will create a Spring Data JPA interface that gives us the standard Spring Data querying capabilities
  - We'll use those methods to query the database.

- **Builds on previous labs**: None
  - We will work in a new project - Lab05.2

- **Approximate Time**: 15-20 minutes

- We will continue to use the Apache Derby database

Lab Preparation - JPA Library / Project

Tasks to Perform

- **Close** all open files and projects
- **Import an existing Maven project** called Lab05.2 into the workspace
  - **File | Import … | Maven | Existing Maven Projects**
  - Click **Next**, Browse to the **workspace\Lab05.2** folder, click **Finish**
  - There is no need to add Java libraries to this project
  - **Maven** will pull in all dependencies needed via *pom.xml*
- Make sure the **Derby database is running**

Define and use a Repository Interface

Lab

Tasks to Perform

- Open **`ItemRepository`** (`com.javatunes.persistence`) to edit
  - Extend the correct interface to make this a Spring Data JPA repository for `MusicItem`
  - That's all you need to do - we inherit the pre-defined query methods (1)
  - Note the query methods in the interface - these are custom query methods that we give you

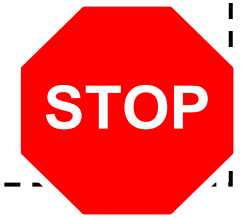
- **Open** `ItemRepositoryTest.java` for editing (*src/**test**/java*)
  - In `com.javatunes.persistence`
  - Autowire an `ItemRepository` instance
    - Where's the implementation? Given to you!
  - Finish the **`testFindPositive()`** to find an item by id.
    - Use the correct JPA interface method - check what it's called.

Test your code

Lab

Tasks to Perform

- **Right click on ItemRepositoryTest** in Package Explorer
  - Select **Run As | JUnit Test**
  - You should get the `MusicItem` with id ==1
  - With no query methods implemented by you!!
- **Experiment** with other Spring Data methods
  - **`size( )`** - existing method of `CrudRepository` interface
  - **`findByArtist()`** - also defined in `ItemRepository`
  - Any others you want to try for a few minutes
- Spring Data is **NICE** !!
  - No more boilerplate queries :-)



LEARNINGPATTERNS

Lab 5.3: Writing Query Methods

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

Lab Synopsis

- **Overview**: In this short lab, we will define our own query method in our repository interface
  - We'll use to to query the database.
- **Builds on previous labs**: Lab05.2
  - Continue working in your **Lab05.2** project
- **Approximate Time**: 10-15 minutes
- We will continue to use the Apache Derby database

Define and use a Repository Interface

Tasks to Perform

- Open **`ItemRepository`** (`com.javatunes.persistence`) to edit
  - Add a method to query for items by artist
  - Remember - you don't have to implement it !

- **Open** `ItemRepositoryTest.java` for editing
  - Find the method **`testFindByArtistPositive()`**
  - Uncomment the **`@Test`** to activate the test
  - Finish it by calling your new query method to initialize the results

- **Run** the tests (Right click, **Run As | JUnit Test**)
  - All tests should pass, and show your results on the console

- Spend a few minutes to try other query methods we provide
  - And if you like, try defining new ones of your own

LEARNINGPATTERNS

Lab 6.1: Spring Transactions

In this lab, we work with Spring's transaction capabilities

Lab Synopsis

- **Overview**: In this lab, we will work with the Spring transaction capabilities
  - We'll configure a transaction manager
  - We'll declare transaction strategies and note the different transactional behavior for different strategies
  - We'll use JPA to provide an environment for TX testing
- **Builds on previous labs**: None
- **Approximate Time**: 35-45 minutes
- NOTE: We've used an expanded naming convention for test methods to make what we're testing clearer (1)

Lab Preparation

- The new root lab directory is: ***workspace\Lab06.1***

Tasks to Perform

- **Close** all open files and projects
- **Import an existing Maven project** called Lab06.1 into the workspace
  - **File | Import … | Maven | Existing Maven Projects**
  - Click **Next**, Browse to the **workspace\Lab06.1** folder, click **Finish**
- Make sure the database is running, and recreate it fresh
  - Running **`dbStart`** (if needed) and definitely run **`dbCreate`**
  - See Lab 5.1 if you need details on using the DB
- We will use our `Catalog` and `ItemRepository` types and the implementations we've seen in previous labs
  - With added methods we can use for testing TX behavior

System Overview

Lab

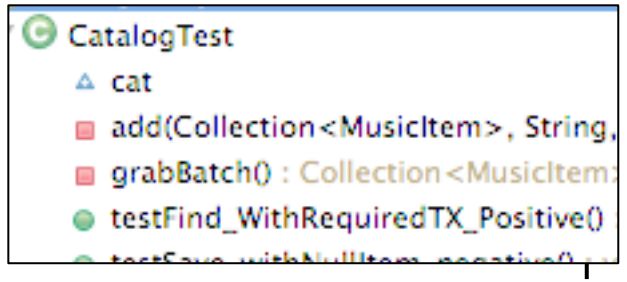
- Repository/Data layer (`com.javatunes.persistence`)
  - JPA-based persistence (`ItemRepository/JpaItemRepository`)
  - Database/JPA configuration done in `SpringRepositoryConfig`
- Service layer (`com.javatunes.service`)
  - Catalog service (`Catalog/CatalogImpl`)
  - Service/repository layers similar to previous labs
- Service layer is transactional
  - With new methods to work with transactions - `persist(MusicItem)`, `persistBatch(Collection<MusicItem>)`, etc.
  - TX configuration done via XML in *transactions.xml*
- `@Configuration` used for bean wiring - it imports the XML config
- Your job is to experiment with how it actually works!
  - We will explore several scenarios, and reconfigure TX settings

Scenario One - Commit

Lab

Tasks to Perform

- Open `CatalogImpl` in `com.javatunes.services`
  - Review the TX settings - various methods are called by different test methods (1)
  - And logging to show TX (*application.properties*)



- **Expand CatalogTest in the Package Explorer**
  - Right-click on method **`testFind_WithRequiredTX_Positive()`** | **Run As JUnit Test**
  - Review the console output - especially the following near the bottom:
    - A new TX was created
    - JDBC connection was acquired, read-only
    - A TX was created for `CatalogImpl` method `findById`, which invoked `JpaItemRepository` method `findOne`, using the same TX
    - Read was done, TX committed
- The “Happy Path” - where everything worked, hence the "positive" test!
  - Does the TX flow you see make sense with our configuration? (2)

Sample Console Output

Lab

- Below is sample console output for `testFind_WithRequiredTX_Positive()` (1)

```
Creating new transaction with name [com.javatunes.service.CatalogImpl.findById]: PROPAGATION_REQUIRED
Opened new EntityManager [SessionImpl(PersistenceContext[entityKeys=[],collectionKeys=[]];ActionQueue
Creating new JDBC DriverManager Connection to [jdbc:derby://localhost:1527/JavaTunesDB]
Setting JDBC Connection [org.apache.derby.client.net.NetConnection@7916f3a] read-only
Exposing JPA transaction as JDBC transaction [org.springframework.orm.jpa.vendor.HibernateJpaDialect
Adding transactional method 'com.javatunes.persistence.JpaItemRepository.findOne' with attribute: PRO
Found thread-bound EntityManager [SessionImpl(PersistenceContext[entityKeys=[],collectionKeys=[]];Act
Participating in existing transaction
```

```
Hibernate: select musicitem0_.id as id1_0_0_, musicitem0_.artist as artist2_0_0_, musicitem0_.musicCategory
```

```
o.s.orm.jpa.JpaTransactionManager        : Initiating transaction commit
```

Scenario Two - Rollback

Lab

Tasks to Perform

- In `CatalogTest` review **`testSave_withNullItem_negative()`**
  - It tries to persist a null object (should blow up)
    - The test catches the appropriate exception
  - Look at the transactional settings for the save( ) method in *CatalogImpl*
- **Expand CatalogTest in the Package Explorer**
  - Right-click on the method | **Run As JUnit Test**
  - The console should show the following near the bottom (see notes)
    - A new TX was created
    - JDBC connection was acquired, autocommit was disabled
    - JPA threw exception received your bogus request, it exploded
    - TX was rolled back
- This scenario illustrates what happens when things go bad
  - JPA does not like it when we ask to persist a null entity!

Scenario Three – NEVER

Lab

Tasks to Perform

- In `CatalogTest` review **`testSize_noTX_positive()`**
  - It gets the catalog size
  - Look at the `size` transactional settings
    - `CatalogImpl size( )` method
    - `JPAItemRepository count( )` method

- **Expand CatalogTest in the Package Explorer**
  - Right-click on the method | **Run As JUnit Test**
  - You will not see any TX creation in the console
  - Though you will see some activity, as the TX subsystem sets the attributes of the annotated methods to NEVER

- This scenario created no transactions at all

Scenario Four - REQUIRED

Lab

Tasks to Perform

- Review `testSaveBatch_withNullEntity_negative()`
  - This scenario passes to catalog a batch of `MusicItems` to save
    - One item in this collection is null
    - This should rollback the entire operation (all items in collection)
  - Look at the `saveBatch` transactional settings in `CatalogImpl`
- **Expand CatalogTest in the Package Explorer**
  - Right-click on the method | **Run As JUnit Test**
  - The console should show the (single) TX rolled back (near the bottom)
  - No items will have been persisted
- Run the ij command line tool via dbSQL (see notes)
  - Execute this query - you should see 20 rows (the initial DB content)

```
select * from MusicItem;
```

Scenario Four - REQUIRES_NEW

Lab

Tasks to Perform

- Locate the save method in **`JpaItemRepository`**
  - It currently has a TX spec of REQUIRED
    - When called by `CatalogImpl.save( )`, it propagates the incoming TX
  - Modify `@Transactional` on this method to **`REQUIRES_NEW`**
  - Same scenario, but each `save()` call now runs in its own NEW TX
    - So any valid item from the collection WILL be committed
- **Expand CatalogTest in the Package Explorer**
  - Run **`testSaveBatch_withNullEntity_negative()`** again
  - The console should show multiple transactions created
    - One for each call to `save()` - one `save()` TX should be rolled back
    - The other `save()` calls should succeed since each is in a separate TX
- Check the contents of the database directly again (via ij)
  - You should see new items in the DB (1)

LEARNINGPATTERNS

Lab 7.1: Spring and the Web

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

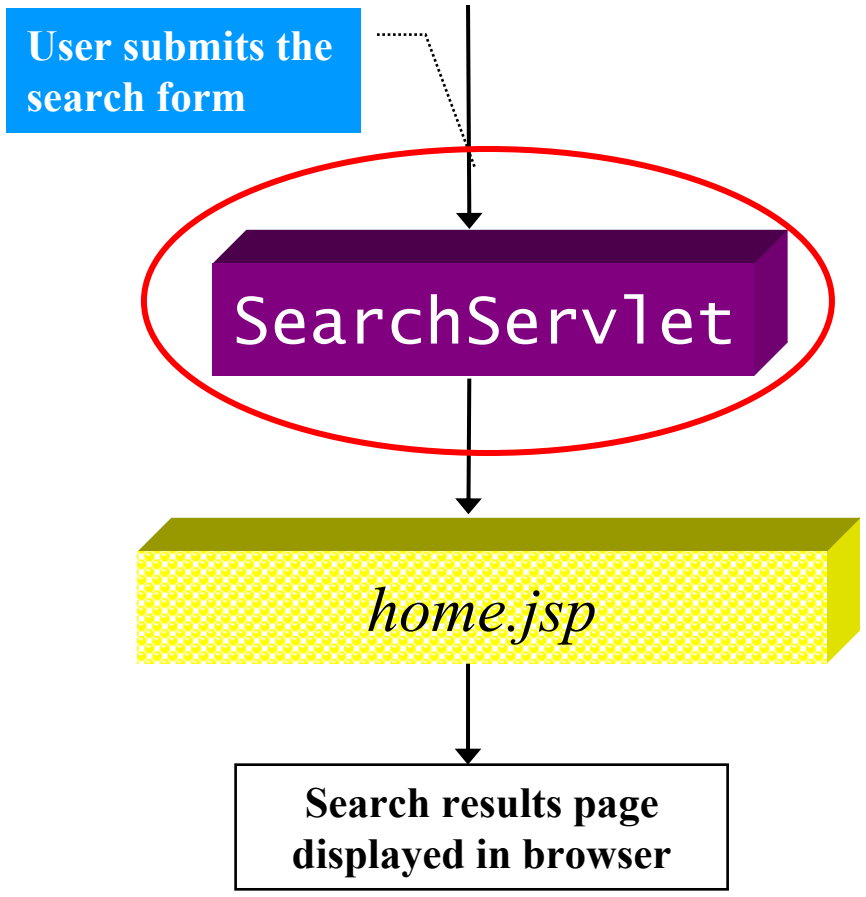
Lab Synopsis

- **Overview**: In this lab, we will integrate the Spring container with a regular Java Web application
  - We will need to set up our Web server (Tomcat)
  - We will then configure the `ContextLoaderListener` in `web.xml`, and use the application context in a servlet
    - We'll look up a Spring bean (a catalog) that's used in a JSP page
  - This is NOT Spring MVC!
- **Builds on previous labs**: none
- **Approximate Time**: 30-40 minutes

The JavaTunes Online Store

Lab

- The Web application we are creating is a small piece of the JavaTunes online music store
  - It displays a search form
  - The search form sends a request to a servlet that does a search and forwards to a results page that displays the results
  - We are working with the servlet processing the search request
  - The flow for JavaTunes appears at right
- The first thing we'll do is set up the environment and the Tomcat Web server



Eclipse and Web Projects

- **Dynamic Web Projects** hold Java Web apps in Eclipse
  - Contain servlets, JSP pages, and static resources (HTML)
  - Web-specific project properties established at project creation
  - *web.xml* has a custom editor for ease of use
- These projects, when used with maven are organized like this:
  - **src/main/java**: Contains all Java source files
  - **src/main/resources**: Other files (e.g. configuration files)
  - **src/main/webapp**: Contains all Web resources
  - **src/main/webapp/WEB-INF**: Same as Java EE WEB-INF
    - Contains ***web.xml***
  - This is different from a standard Eclipse Web project
  - Also different from a standard Java EE Web app
    - But when deployed, a standard WAR is build

Lab Preparation

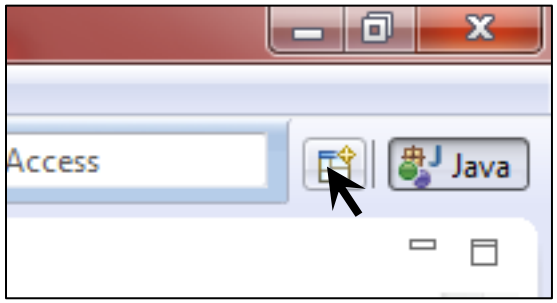
Lab

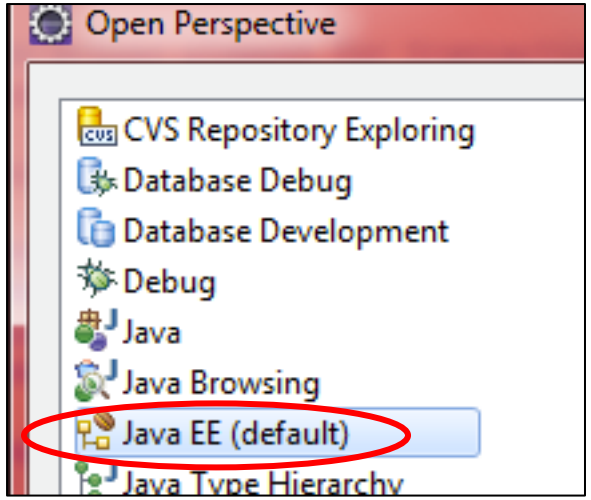
- The root lab directory where you will do all your work is:

  workspace\Lab07.1

Tasks to Perform

- **Close** all open files and projects
- Open the Java EE perspective, by clicking the Perspective icon at the top right of the Workbench, and select **Java EE**
  - This perspective is well suited for these labs





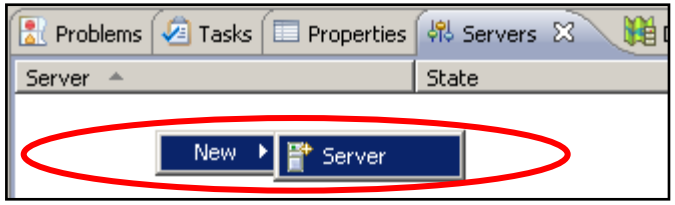
Creating a Server

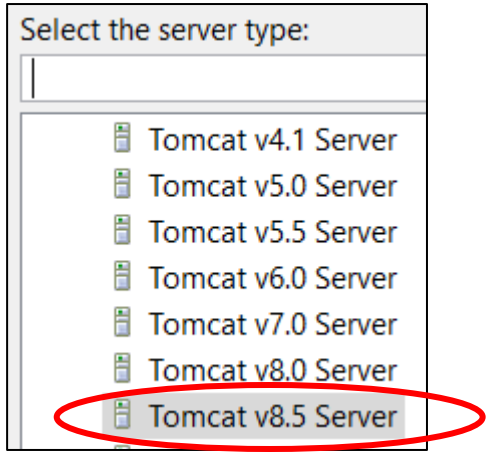
Lab

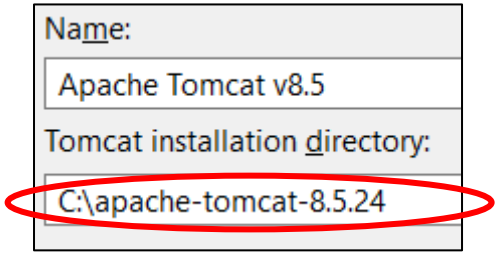
- We will use Tomcat to run our Web applications - first we need to create a server in Eclipse *

Tasks to Perform

1. Go to the Servers view, right click, and select **New | Server**
2. In the next dialog, select **Apache | Tomcat V8.5**(1) and click **Next**
3. In the next dialog, browse to your **Tomcat install directory**(2), click **OK**, and then **Finish**(3)







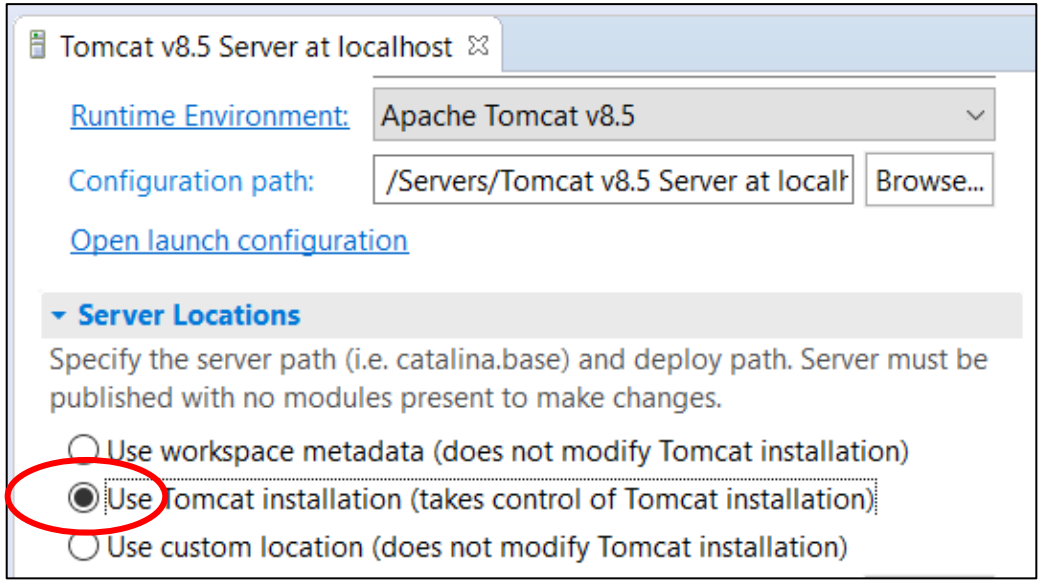
Configure Server

Lab

- We will reconfigure the deploy location for our server
  - This will help prevent Eclipse deploy issues we've seen (see notes)

Tasks to Perform

- In **Servers view**, double click on the Tomcat server
  - The server configuration should open in the editor pane
  - Check the "Use Tomcat installation" choice (see below)
  - Save the configuration (See notes about OutOfMemory Exception)

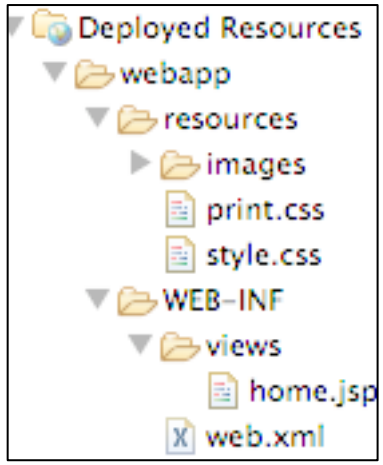


Lab Preparation - Bring in Project

Lab

Tasks to Perform

- **Close** all open files and projects
- **Import an existing Maven project** called Lab07.1
  - **File | Import … | Maven | Existing Maven Projects**
  - Click **Next**, Browse to the **workspace\Lab07.1** folder, click **Finish**
- This creates a (maven-based) Web project in Eclipse
  - With the structure described previously
  - In Package Explorer, webapps folder is visible under *Deployed Resources*
  - The server runtime is also set up differently compared to a normal Eclipse Web Project (see notes)



Review pom.xml

Lab

- Maven will automatically add the appropriate .jars to the application based on pom.xml

Tasks to Perform

- Open *Lab07.1/pom.xml*. **There are no changes** needed here.
  - Select the pom.xml tab at the bottom of the editor to view XML
- Review the dependencies you see specified, including:
  - **spring-boot-starter**
  - **spring-boot-starter-web**
  - **spring-context**
  - **jstl**
- Select the **Dependencies** tab to see the dependency versions
- Select the **Dependency Hierarchy** tab to see all the associated .jar files that each dependency pulls in
  - That's the power of Maven plus Spring Boot…it pulls those dependencies in for you

Configure ContextLoaderListener

Lab

Tasks to Perform

- Open up *WEB-INF\web.xml* for editing
  - Find the **<context-param>** with param-name of **contextClass**
  - Set the param-value (with no spaces or newlines) to

```
org.springframework.web.context.support.
        AnnotationConfigWebApplicationContext
```

  - Find the **<context-param>** with param-name of **contextConfigLocation**
  - Set the param-value to:

```
com.javatunes.config.SpringConfig
```

  - Locate the **<listener>** element with class="TODO" and change the value to:

```
org.springframework.web.context.ContextLoaderListener
```

- You've integrated your Web app with Spring! (1)

Finish SearchServlet

Lab

Tasks to Perform

- Open `com.javatunes.web.SearchServlet` for editing
  - Notice the Spring imports for the web context classes
  - Look for the TODO comments
  - Get the Spring context via the helper class (See manual slides)
  - Get a catalog from the context (like earlier labs)
  - Note how they are put on the servlet request (already done)

- You have just added Spring to a simple webapp!
  - Think about it - You get everything
  - DI, Transactions, AOP, Etc
  - All by adding the listener to your webapp!

Deploying an Application

Lab

Tasks to Perform

- **Remove** any existing project from the server
  - Right click on server in Servers view, select **Add and Remove Projects**, and remove existing projects
  - **Or** (easier)
    - Select project under the server node in Servers view, type **Delete**
- **Add** new project to the server
  - In the same dialog as above, select your project in the left hand pane
    - Click **Add**
  - **Or** (easier)
    - **Drag** project from the Project Explorer onto server in Servers view
- **Restart** the server (**Right** click on it, **Restart** - or start if not started)
  - **NOTE**: You always need to **restart the server after code changes** (1)

Viewing the Web Application

Lab

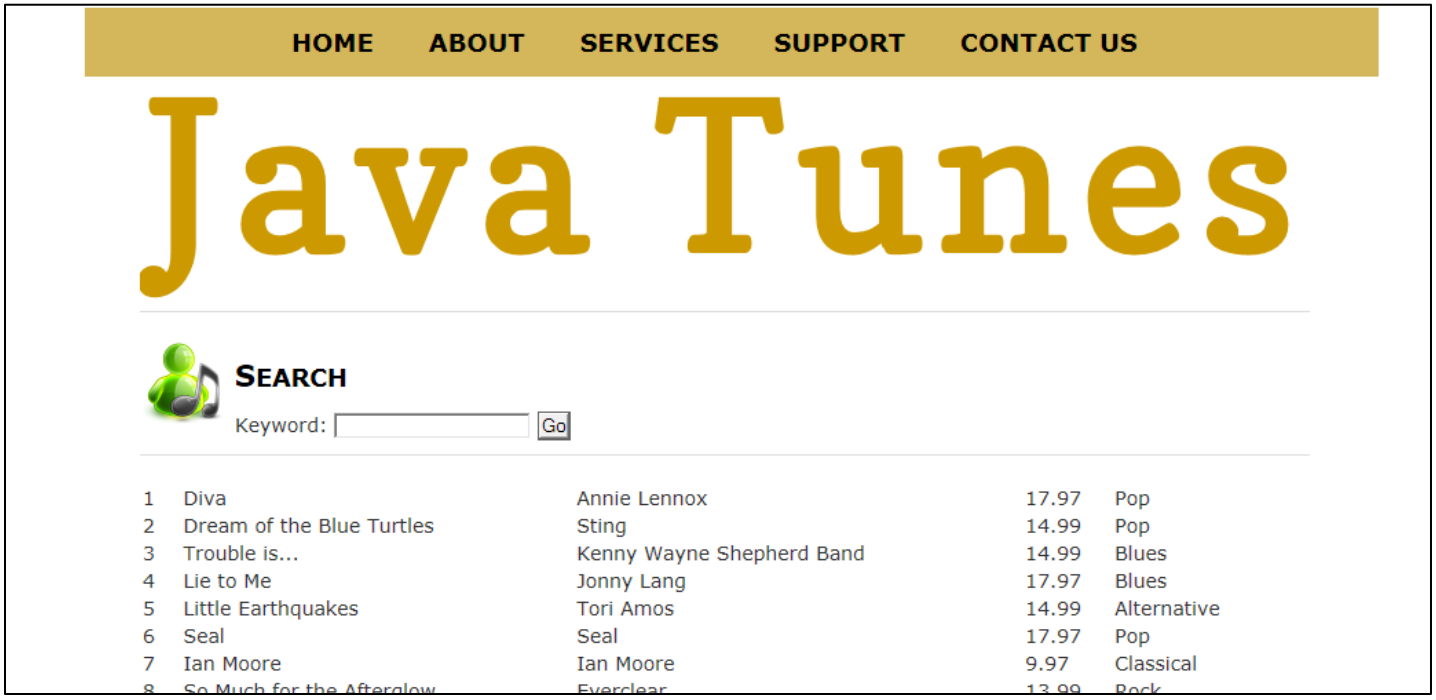
- Open an external browser (1) onto the Web app
  - At: *http://localhost:8080/javatunes*
- Note if using IE: It **can be misleading** as it caches pages, and it's hard to clear the cache
  - If you ever feel your having browser issues, shut it down and restart
  - Or try a different browser

Test Your Application

Lab

Tasks to Perform

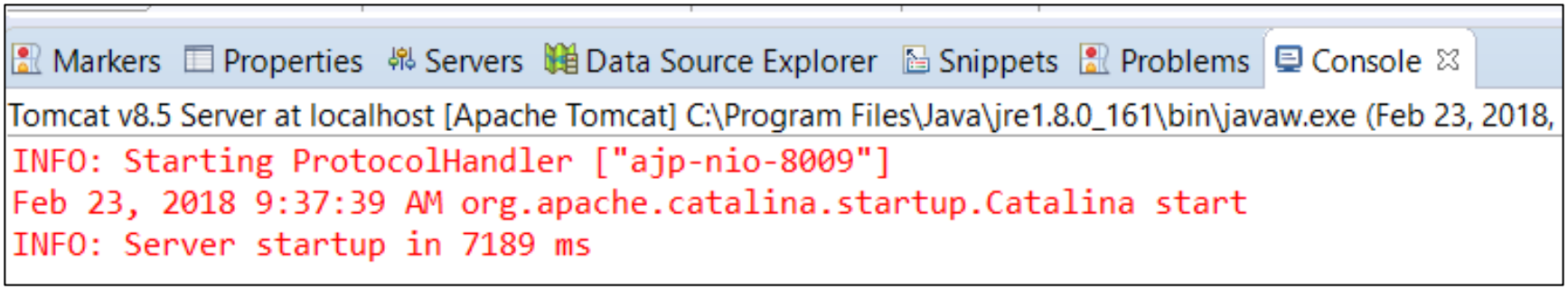
- The **javatunes** web app, displays a search form - so type in the letter "a" to search on and submit the form
  - The `SearchServlet` will use your Spring beans to do the search, and then forward to a JSP to display the results
  - That's it - you're using Spring in a Web application



Server Startup

Lab

- You can open the **Console** view to see output from the server startup and web app deployment
  - This can be useful to look for exception stack traces
  - You can also look at the server status in the **Servers** view



LEARNINGPATTERNS

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

Lab Synopsis

- **Overview**: In this lab, we'll set up Spring MVC, and use some basic capabilities to configure a simple controller
  - We'll first set up the `DispatcherServlet` to handle web requests
  - Next we'll configure a very simple controller to handle a request
  - We'll run our application to see these pieces at work, and in future labs we'll enhance it to use more Spring MVC capability
- **Builds on previous labs**: none
- **Approximate Time**: 25-35 minutes

Lab Preparation

- The (new) root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab07.2

Tasks to Perform

- **Close** all open files and projects

- **Import an existing Maven project** called Lab07.2
  - **File | Import … | Maven | Existing Maven Projects**
  - Click **Next**, Browse to the **workspace\Lab07.2** folder, click **Finish**

- Maven will configure and provide all necessary .jar files
  - Review pom.xml under the project root if you are interested.

Configure DispatcherServlet

Lab

Tasks to Perform

- Open `com.javatunes.config.JavaTunesWebAppInitializer`
  - Note how it extends Spring's class for Servlet-3 initialization
- Locate the `getServletMappings()` method
  - Finish this method using a URL pattern of **`/content/*`**
- Locate the `getRootConfigClasses()` method
  - Set the root config to be `SpringConfig` (which we've used before)
- Locate the `getServletConfigClasses()` method
  - Set the root config to be `WebConfig` (which will configure Spring MVC)
- Save the file
- Open `com.javatunes.config.WebConfig`
  - Add the annotation to enable Spring MVC
  - Add the annotation to specify auto-scan of **`com.javatunes.web`** (1)
  - Save the file

Finish the Controller

Lab

Tasks to Perform

- **Open** `com.javatunes.web.HomeController` for editing
  - It should handle HTTP GET requests to */home* (1)
- Add annotations to **`HomeController`** to declare it a controller class, and to map it to the request `/home`
  - You will need two annotations
- Add annotations to **`get()`** so that it handles an HTTP GET request
  - Not how it returns a simple string in <h1></h1> tags
  - We'll use a JSP page in the next lab
- Open *web.xml* for review (**Nothing** you need to do here)
  - Note how it sets a welcome-file of content/home
  - This will be handled by our Spring MVC controller

Deploying an Application

Lab

Tasks to Perform

- **Remove** any existing project from the server
  - Right click on server in Servers view, select **Add and Remove Projects**, and remove existing projects
  - **Or** (easier)
    - Select the project under the server node in Servers view, type **Delete**
- **Add** this project to the server
  - In the same dialog as above, select your project in the left hand pane
    - Click **Add**
  - **Or** (easier)
    - **Drag** the project from the Project Explorer onto the server in Servers view
- **Restart** the server (**Right** click on it, **Restart** - or start if not started)

Test Your Results

Lab

Tasks to Perform

- Open a browser to: *http://localhost:<8080>/javatunes*
  - Which should serve your welcome page of content/home - and display the content from your controller (1)
  - This shows that your controller class is being used

- Explanation: When the request for content/home is made, the dispatcher servlet receives this request (due to the servlet mapping)
  - It's dispatched to `HomeController.get()` based on its configuration
  - `get()`'s return value is used as the view (because of `@ResponseBody`)
- You can see how simple it is to build a controller using Spring MVC
- Note: **Nothing else is implemented**
  - We'll add searching shortly

LEARNINGPATTERNS

Lab 7.3: View Resolvers

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

Lab 7.3: View Resolvers

- **Overview**: In this lab, we'll add a view resolver to our configuration
  - The controller will now use a logical view name as its response
- **Builds on previous labs**: Lab 7.2
  - Continue working in your Lab07.2 project
- **Approximate Time**: 15-20 minutes

Add a View Resolver, Finish Controller

Tasks to Perform

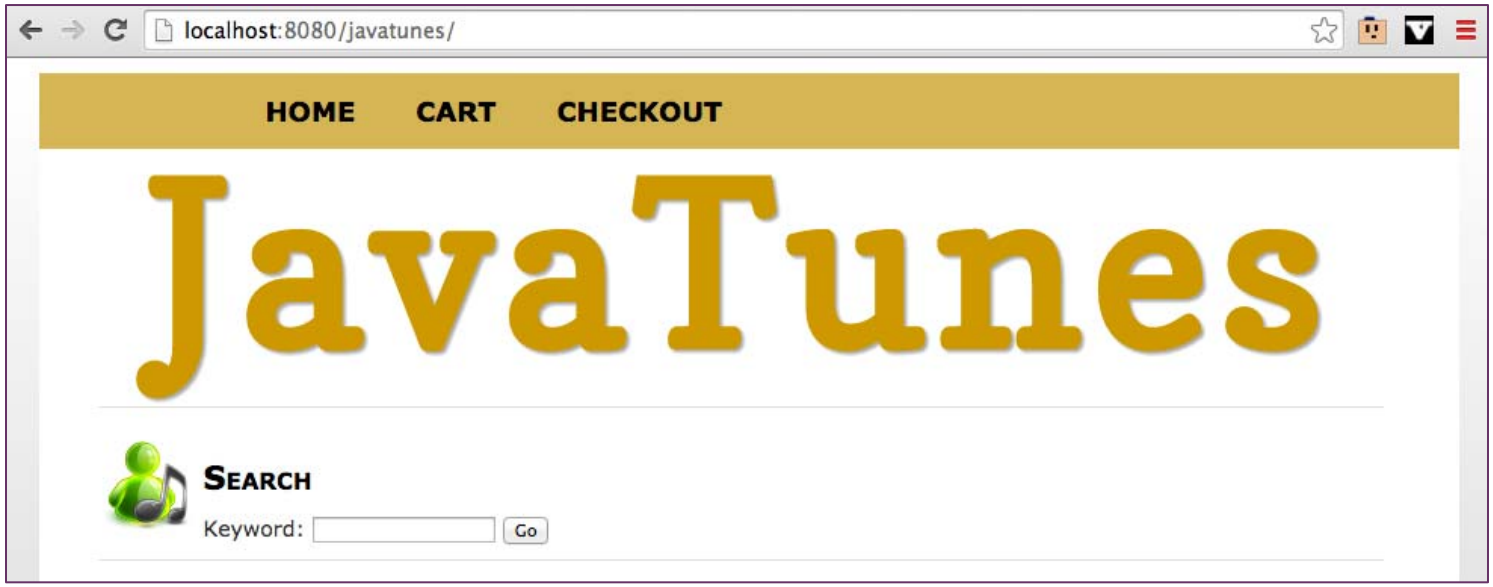
- **Open** `com.javatunes.config.WebConfig` for editing
  - Have the class implement `WebMvcConfigurer` (1)
  - Add a bean definition that creates, configures and returns an `InternalResourceViewResolver` bean (see the manual slides)
  - Configure a prefix of **`/WEB-INF/views/`**, and a suffix of **`.jsp`**
  - e.g. to map the logical view name `logon` to */views/logon.jsp*
- **Open** `com.javatunes.web.HomeController` for editing
  - Modify the `get()` method to return a logical view name of "home"
    - This will end up using ***/WEB-INF/views/home.jsp***, which we supply
  - **Remove** the **`@ResponseBody`** annotation (2)
- Open up *web.xml* for review, and note the welcome-file
  - It's in *Deployed Resources/webapp/WEB-INF*
  - The welcome file (default web page) was configured as a URL handled by the dispatcher servlet (**`content/home`**)

Test the App

Lab

Tasks to Perform

- **Restart** the server (In Eclipse, right click on server, **Restart**)
- Test your Web app by browsing to *http://localhost:8080/javatunes*
  - You should see a web page displayed
  - Try *localhost:8080/javatunes/**content/home*** - should show same page
  - NOTE: Searching **does not work** yet



[Optional] Handle Root of Web App

Lab

Tasks to Perform

- Optionally, you can try having the dispatcher handle the app root
  - **Make a copy of the project** and work in the copy (1)
- In `JavaTunesWebAppInitializer`, set the dispatcher servlet mapping to /
- In `WebConfig`, enable the default servlet handling by writing the `configureDefaultServletHandling` method
  - Refer to the session slide on "Mapping DispatcherServlet to /"
- In `HomeController`, add a RequestMapping URL of /
  - Keep the /home also
- **Deploy** your project (2), **restart** the server again, test by browsing to *http://localhost:8080/**javatunes/home***
  - You should see the same page displayed
- Undeploy this project, redeploy the non-optional one (3)
  - Future labs assume you're using the **non-optional** version

LEARNINGPATTERNS

Lab 7.4: Client Input / Model Data

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

Lab Synopsis

- **Overview**: In this lab, we'll extend the functionality of our web app to do a search for music items
  - We'll extract a request parameter with `@RequestParam` to get the search keyword
  - We'll use `ModelAndView` to return a model object containing results that's used to generate the resulting view
- **Builds on previous labs**: Lab 7.3
  - Continue working in your Lab07.2 project
- **Approximate Time**: 20-30 minutes

Finish the Controller

Lab

Tasks to Perform

- **Open** `com.javatunes.web.HomeController` for editing
- Inject a Catalog bean into `HomeController` using `@Autowired`
  - This is our familiar catalog type - already configured for you
  - Just declare a field inject with `@Autowired` (1)
- Add a new method, `processSearch()` to the controller
  - Specify a request mapping with HTTP POST, and handle a request URL of **`home/search`** (2)
  - Provide a `String` argument for the keyword
    - Initialize this with a request parameter for "keyword" - the field name passed from the form
  - Use `Catalog.findByKeyword` to search on the keyword
  - Return a `ModelAndView` object that
    - Specifies a view of ***home***
    - Includes a model attribute with the name **`matches`** and the search results

Finish the JSP

Lab

Tasks to Perform

- **Open** *WEB-INF/views/home.jsp* and find the search form in it
  - Search for the TODO (element starts with `<form`)
- Set the HTTP method and action so the form submission is handled by `processSearch()` in our controller
  - This is **not fully set** in the JSP page provided in the setup
  - The method should already be set to **post**
  - What should the URL for the action be to get to the `processSearch()` method - fill this in
- Still in home.jsp, find the **`c:forEach`** (which displays search results)
  - You can search for TODO
  - Finish it, so the `forEach` iterates **over the correct model object**
    - What's the name that processSearch uses for this? Use the same one

Run It

Lab

Tasks to Perform

- **Restart** the server so your changes are picked up
  - Right click on the server, select **Restart**
- Browse back to: *http://localhost:<8080>/javatunes*
  - In the Search box on that page, type in the letter "a" and submit
  - DispatcherServlet invokes `HomeController.processSearch()`
  - The controller uses a catalog to search
  - It returns a `ModelAndView` instance initialized with the collection (with name "matches") and a view ("home")
  - The `DispatcherSerlvet` gets the model data out of the `ModelAndView` instance
  - It puts it on the request with the name "matches"
  - It forwards to the view
- You should see your search results displayed

[Optional] Use Model Object

Lab

Tasks to Perform

- Optional: Change `processSearch()` to take a `Model` object
  - Comment out your old method definition, and create a new one
  - The `Model` object should be the second argument to the method
  - The method should return a string instead of a `ModelAndView`
  - The functionality should be the same - it should add the result into the model with the key "matches" and the view should be *home.jsp*
- **Restart** the server
- Test the Web app again
  - It should work exactly as before

LEARNINGPATTERNS

Lab 8.1: Forms and Model Objects

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

Lab Synopsis

- **Overview**: In this lab, we'll expand on the Spring MVC functionality that we use
  - We'll use the Spring form tag library, and an associated model object to handle the form submission
  - We'll bind model attributes to request parameters
  - We'll modify our controller to use the model objects
  - This will be used in our JavaTunes web app to handle the search functionality
- **Builds on previous labs**: none
- **Approximate Time**: 25-35 minutes

Lab Preparation

Lab

- The (new) root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab08.1

Tasks to Perform

- Remove previous project from server
- **Close** all open files and projects
- **Import an existing Maven project** called Lab08.1
  - **File | Import … | Maven | Existing Maven Projects**
  - Click **Next**, Browse to the **workspace\Lab08.1** folder, click **Finish**
- Maven will configure and provide all necessary .jar files
  - Review pom.xml under the project root if you are interested.

Finish the Controller

Tasks to Perform

- Open `com.javatunes.web.Search` for review
  - This is a model class with three properties – a keyword (`String`), category (`String`) and search results (`Collection<MusicItem>`)
- Open `com.javatunes.web.HomeController` for editing
  - Note how `get()` and `processSearch()` both have parameters of type Search
  - Annotate the parameters in both methods with `ModelAttribute` so that they are bound to a model attribute named "search"
- Finish `get()` by initializing the keyword to "`Diva`"
  - This value will show up in the initial rendering of the form
- Finish `processSearch()` as follows:
  - Extract the keyword from the search object, and use it to search
  - Add the matches into the search object

Finish the JSP Pages

Lab

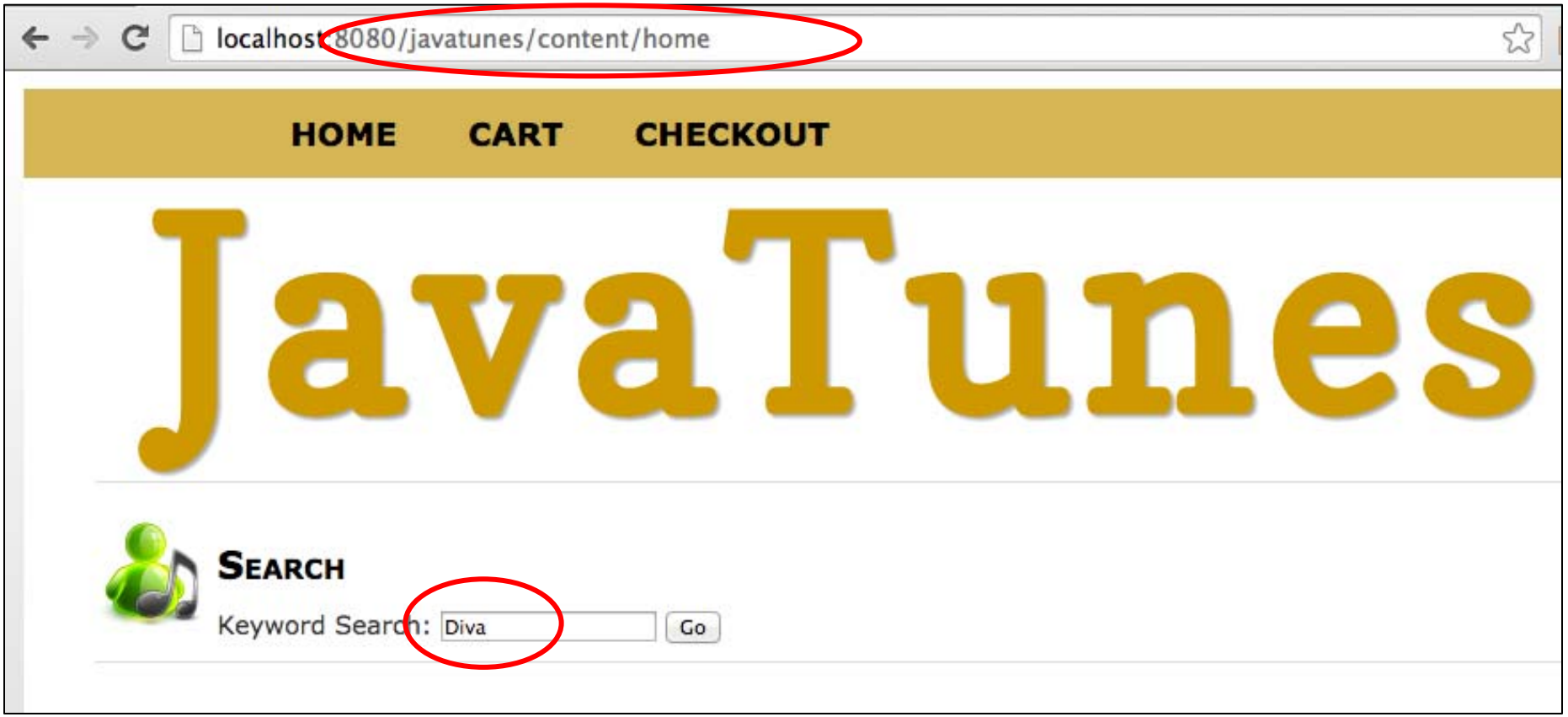
Tasks to Perform

- Open *WEB-INF/views/home.jsp* (it contains the search form)
  - Note the taglib for the Spring MVC form tag library at the top
  - **Modify the opening `form` tag** to include the form tag library prefix, and to bind to the model attribute with a name of "`search`"
  - **Modify the closing `form` tag** to include the form tag library prefix
  - Finish the `form:input` tag's path attribute to bind to the "keyword" property of the model object
- Continue working in *home.jsp*
  - The search command bean is available in the results view also
  - Search for the `c:forEach` that's a few lines below the form
  - Finish the `c:forEach` tag's items attribute by initializing it to refer to the results property of the search bean

Form Initialized and Rendered

Lab

- When you run it (next) you'll see something like the below
  - We've browsed to the home page, and the response to that request has been generated by the form it contains
  - Note how the keyword field has been initialized to "Diva"



Test Your Application

Lab

Tasks to Perform

- Deploy to the server as you did earlier (restart the server)
- Open a browser on: *http://localhost:<8080>/javatunes*
  - Click the link to go to the search form – you should see your default search term of "Diva"
    - This is populated by the model that's initialized in `HomeController.get()`
  - Do a search – you should see familiar search results
  - Congratulations - you've made a simple, but complete, Spring MVC app
  - See optional part on next slide

[Optional] Add Reference Data

Tasks to Perform

- **Optional**: Add reference model data to the application
- In `HomeController`, annotate `populateCategories()` so it adds an attribute named "`categories`" into the model
- In `HomeController`, we'll search by category if the incoming keyword is null or empty
  - Add code to `processSearch()` to test the keyword property to see if it's null, or of zero length
  - If it is, then search by the category instead (which is also a property in the class - and should have been populated from the form)
- *home.jsp*, uncomment/finish `form:select` to bind to the **`category`** property - initialize the items from the **`categories`** attribute
- **Restart** / run the app again and test this functionality
  - Make sure to test **with no keyword in the search field**
  - This will trigger the search by category

LEARNINGPATTERNS

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

Lab Synopsis

- **Overview**: In this lab, we'll access the HTTP session from Spring controllers
  - We'll add cart functionality, and store the cart on the session
  - We'll store a model object (our Search bean) on the session also
    - So we can easily search from the Cart page
    - Using `@SessionAttributes`
  - You'll do a lot of the Spring MVC work in the cart

- **Builds on previous labs**: none

- **Approximate Time**: 45-60 minutes

Lab Preparation

Lab

- The (new) root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab08.2

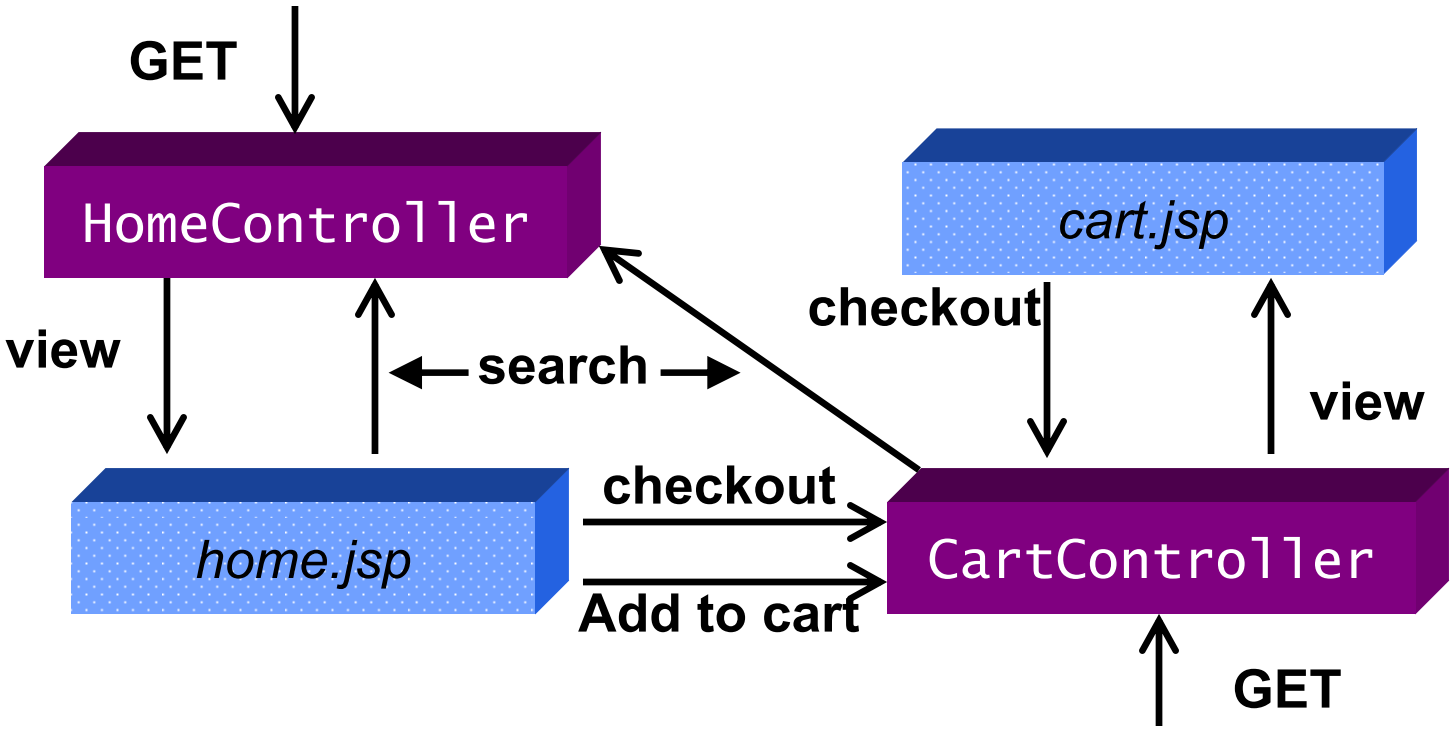
Tasks to Perform

- Remove previous project from server
- **Close** all open files and projects
- **Import an existing Maven project** called Lab08.2
  - **File | Import … | Maven | Existing Maven Projects**
  - Click **Next**, Browse to the **workspace\Lab08.2** folder, click **Finish**
- Maven will configure and provide all necessary .jar files
  - Review *pom.xml* under the project root if you are interested.

Complete Web Application Flow

Lab

- ***home.jsp***: Displays the home page (same as earlier labs)
- ***cart.jsp***: Displays a shopping cart (plus our search form)
- **`HomeController`**: As seen previously
- **`CartController`**: Process cart-related requests



Search Form Considerations

Lab

- The search form appears in both Web pages
  - And it uses a Search model bean
  - We want the search term to be kept if we move between pages
  - How? We'll put the Search model bean on the session

- Another consideration: The search form is present in both the:
  - **Home page** (served by GET on `HomeController` or a search)
  - **Cart page** (served by GET on `CartController` or Add-to-cart)
  - So each controller has to be prepared to initialize the search model bean since it may be the first one accessed in a session
  - We'll do this by adding in the search model bean as reference data
  - i.e. in a method annotated with `@ModelAttribute` that creates it
  - We'll add the cart (an `ArrayList`) to the session in the same way

Review View Pages

Lab

Tasks to Perform

- Open *WEB-INF\views\home.jsp* and *cart.jsp* for review
  - We've completed these for you
- In *home.jsp*:
  - Note the top navigation links for Cart and Checkout - they will now make requests handled by a `CartController`
  - Note the search form - which has not changed
  - Note the search results display - it now generates a link going for each item that goes to */cart/content/add* (to add an item to the cart)
- In *cart.jsp*
  - Note the same top navigation links
  - Note the search form
  - Note the display of the cart - a `forEach` over the "cart" bean
    - This bean will be added (to the session) in `CartController`

CartController (1 of 3)

Lab

Tasks to Perform

- Open `CartController` for editing and do the following:
- Annotate the class to declare it as a controller, and map it to the request `/content/cart`
  - Review `HomeController` if you need a refresher on how to do this
- The `get()` method generates the page - finish it as follows:
  - Add an annotation so it handles an HTTP GET request
  - Add an annotation so the **Search parameter** in the method is bound to a model attribute named "`search`"
  - Return an appropriate value so that ***cart.jsp*** will be the view
  - See Lab 8.1 if you need a refresher on how all this is done

CartController (2 of 3)

Lab

Tasks to Perform

- **`add()`** adds an item to the cart - finish it as follows
  - Annotate it to handle GET requests for `/content/cart/add`
    - We'll be accessing it via regular links - which generate a GET
  - Add an annotation so the **`cart` parameter** in the method is bound to a model attribute named "`cart`"
  - Initialize the `id` argument to add() from an "id" parameter on the request (make it required also)
    - Hint: We did this similarly in Lab 7.4 `processSearch()` method
  - Return an appropriate value to return to the home page (***home.jsp***)
  - Take a minute to view the logic in this method
    - It's standard HTTP session stuff - not dependent on Spring MVC
    - Note the HTTP session method parameter - that is initialized automatically when the controller method is called

CartController (3 of 3)

Lab

Tasks to Perform

- **`checkout()`** "empties" the cart - finish it as follows
  - Annotate to handle GET requests for `/content/cart/checkout`
  - Add a **`SessionStatus` parameter** to the method
  - Within `checkout()`, use the `SessionStatus` instance to indicate that use of the session is complete
  - Note how the return value does a redirect - this makes sure the session is set up properly
    - Nothing needs be done with the return value - it is complete
    - It will redirect to the cart page
  - When checkout is called, the cart will now be removed from the session
    - We don't bother writing the cart processing for checkout - it's not relevant to the web flow we're working on now

Set up Model Beans In the Session

Lab

Tasks to Perform

- Continuing in **CartController**
  - Add code so it sets up the search and cart beans as follows:
  - Find **createSearch()** - it creates/returns a Search bean
    - Annotate this method to put the bean in the model with the name "search"
  - Find **createCart()** - it creates/returns the cart (an ArrayList)
    - Annotate this method to put the cart in the model with the name "cart"
  - **Annotate CartController** to store the **"cart"** and **"search"** beans on the session (@SessionAttributes)

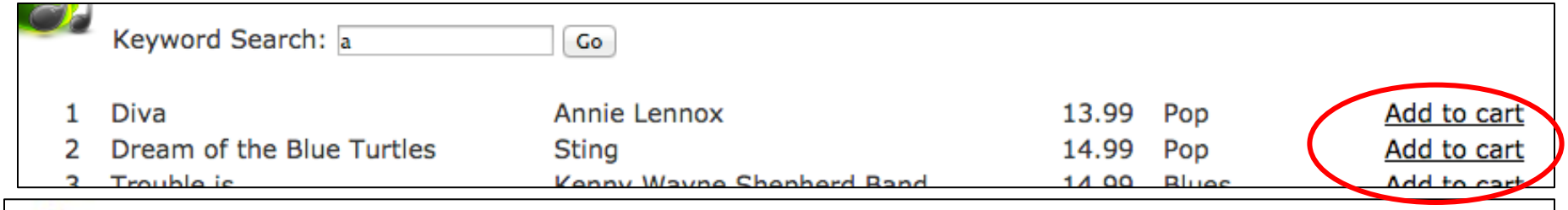
- Open **HomeController** for editing (it also sets up the search bean)
  - Find **createSearch()** and annotate it to put the returned bean in the model with name "search"
  - **Annotate HomeController** to store the **"search"** bean on the session

Test Your Web App

Lab

Tasks to Perform

- Deploy this project to the server (**restart** the server)
- Open a browser on: *http://localhost:<8080>/javatunes*
  - You should see links on the results to add items to the cart (below)
  - Add items to the cart, then click the **Cart** link to view the cart
  - You should see the cart display (bottom)



Recap

Lab

- We've done a lot in this lab
- Completely set up the `CartController` to be a Spring MVC controller
  - Using a lot of the learning we've had previously
- Set up model data to live on a session
- You've come a long way! Be happy!

LEARNINGPATTERNS

Lab 9.1: A Simple REST Resource

Create a RESTful Resource Using Spring MVC

Lab Synopsis

- **Overview**: In this lab, we will become more familiar with REST by creating a simple REST resource
  - We'll use URL Templates and `@RequestMapping` to map request URIs to Spring MVC controllers
  - We'll use `@RestController` to indicate that all handlers return response data
  - We'll simply access the resource from a browser to test it (for now)
- **Builds on previous labs**: none
- **Approximate Time**: 25-35 minutes

Lab Preparation

- The (new) root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab09.1

Tasks to Perform

- **Remove** previous labs from the server, **close** open files/projects (1)
- Import an existing Maven project called **Lab09.1**
  - See earlier lab instructions if you need more detail on this
- Open and briefly review the overall configuration
  - Mostly in `com.javatunes.config` classes
  - Some in *web.xml*
  - It should be familiar from our regular Spring MVC configuration

Finish the Resource Controller

Tasks to Perform

- Open **`ItemsResource`** (in package `com.javatunes.rest`)
  - Review the class briefly
  - It contains an injected `Catalog` object
  - It declares two methods we'll use as REST handlers

- Add annotations to accomplish the following
  - Indicate the class is a REST controller
    - So all methods have an implicit `@ResponseBody`
  - Map the entire class to the **`/items`** URI
  - Configure **`getAllItems()`** as a controller, also mapped to **`/items`**
  - Configure **`findItemById()`** as a controller, mapped to a URI of the form **`/items/12`**, using a **`URI Template`** with a name of **`id`**
    - Add an annotation to bind the URI Template variable to the "id" parameter that is already present in the method

Deploy and Test

Lab

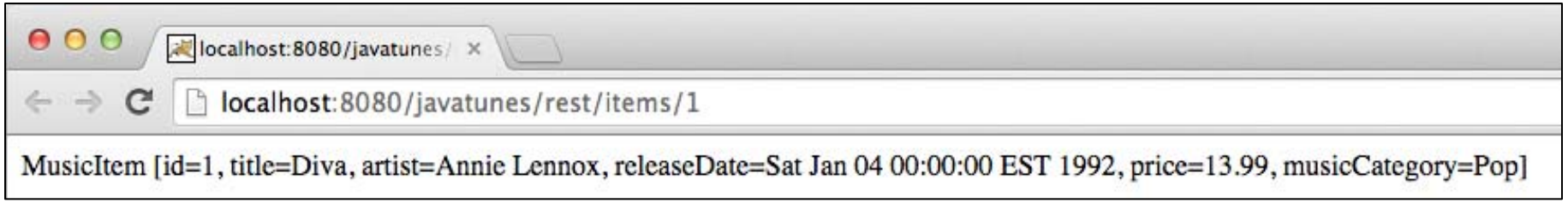
Tasks to Perform

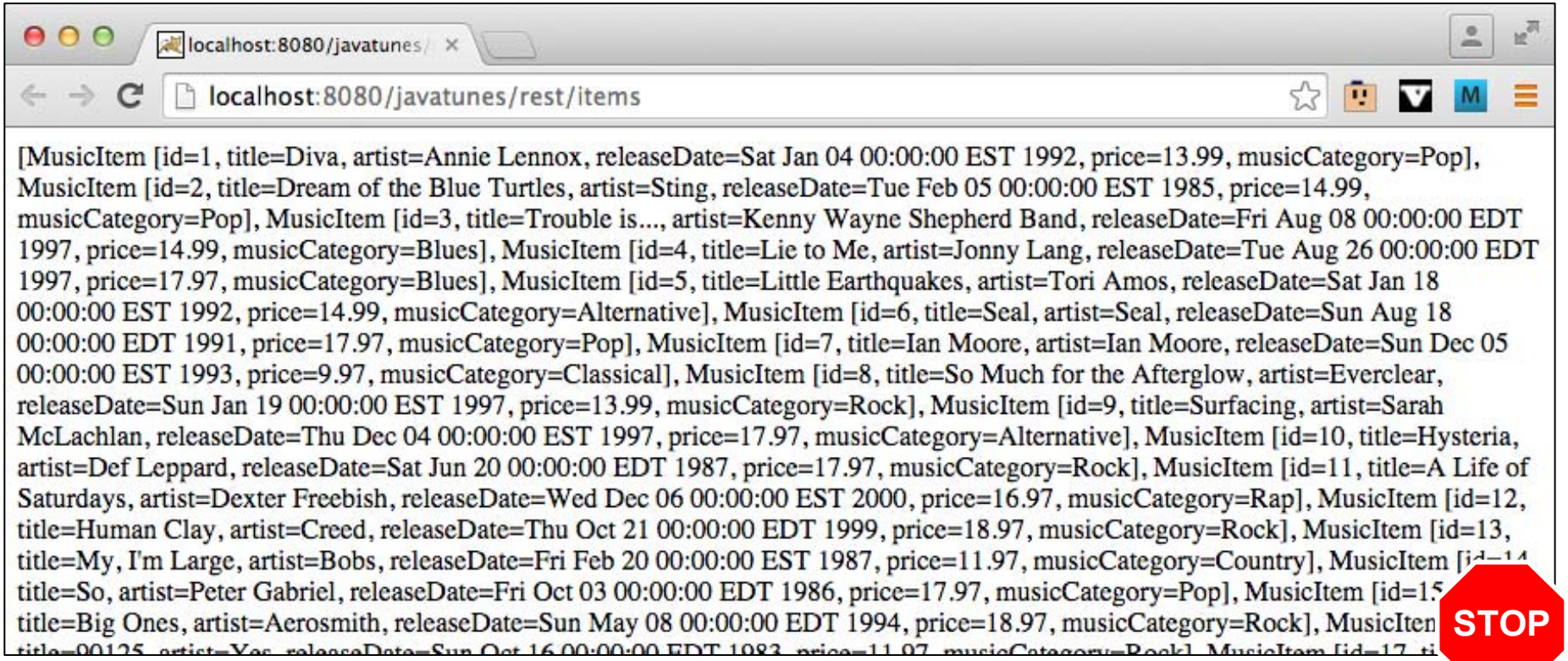
- Deploy to the server as usual (restart the server)
- Browse to ***http://localhost:8080/javatunes***
  - The home page has links to our two REST resources (all under `/rest`)
- Click the link for one item: ***/items/1***
  - Should show the string representation of the item with id == 1
  - This is handled by `findItem(Long id)`
  - The id parameter is initialized from the URI template in the request
- Go back, and click the link for all items: ***/items***
  - Should show the string representation of all the items
  - This is handled by `getAllItems()`
  - Note: We're using a browser as a client for simplicity in this lab
- It's easy to write and invoke REST services, as you can see

Results

Lab

- Below are screen shots for a single item, and all items (bottom)





LEARNINGPATTERNS

Lab 9.2: Use Ajax in a Client

Access REST data from an Ajax client (a Web page)

Lab Synopsis

- **Overview**: In this lab, we'll use Ajax to access your REST resource
  - Via a jQuery call from a page viewed in a browser
  - We supply the needed jQuery code
- **Builds on previous labs**: Lab 9.1
  - Continue working in your Lab09.1 project
- **Approximate Time**: 25-35 minutes

Access REST via Ajax

Lab

Tasks to Perform

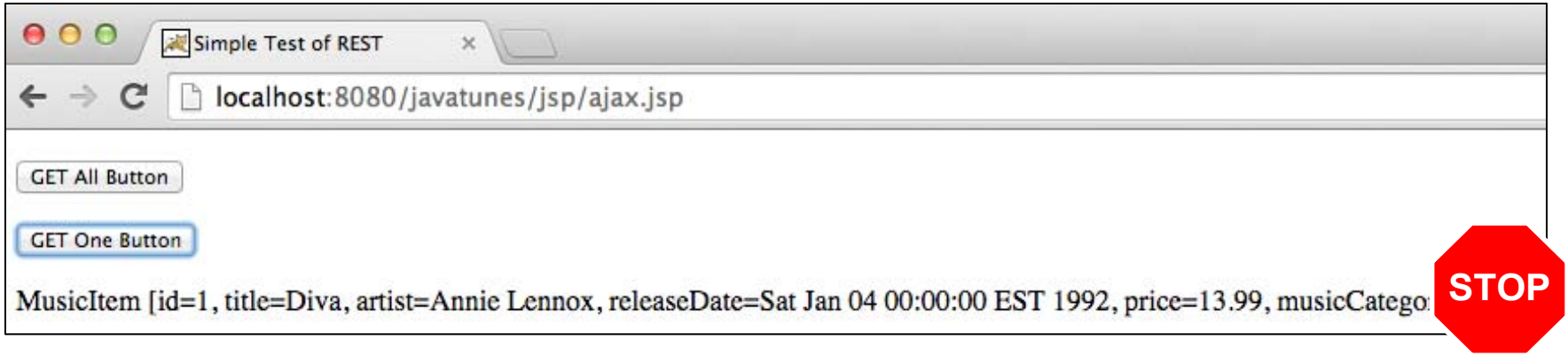
- Open *webapp/**views/ajax.jsp*** for editing
  - Note that we have two buttons there, with ids of **`getAllButton`** and **`getOneButton`**, and a div with id of **`ajaxContent`**
  - Note also that we include the jQuery library from the Google CDN (1)
  - These provide easy Ajax functionality to access our REST resources
- We provide click handlers for both buttons
  - But you need to **add the REST URI** to access our REST resources
  - Search for the string **TODO**
  - In the click handler for all items (`$('#getAllButton').click`) set the **`url`** to that of our REST resource returning all items
  - In the click handler for one item (`$('#getOneButton').click`) set the **`url`** to that of our REST resource for a single item with an id of 1
    - Don't change the JSP EL expression for the context root, that looks like this: `${pageContext.request.contextPath}`

Use Ajax to Access REST Resources

Lab

Tasks to Perform

- Restart the server
- Browse to your web app in an **external browser** (not the internal Eclipse browser) (1)
  - On the home page for the app, click the link to "Try some Ajax"
  - Try out your buttons - when pressed, they should get the REST data via an Ajax call and update the div in the page
  - The complete page is not updated - only the div containing the data
  - That's the power of Ajax, and it's well served by REST data
  - Note: The "Try some Ajax with JSON" page **won't work yet**



LEARNINGPATTERNS

Lab 10.1: A JSON Resource

Create a RESTful Resource Returning JSON Data

Lab Synopsis

- **Overview**: In this lab, we will work with the JSON support in Spring MVC/REST
  - We'll modify our REST resource to support the return of JSON data
  - We'll use the Jackson libraries for our JSON support
    - Spring Boot pulls these in for us
  - We'll access the resource and display the data from an Ajax page
- **Builds on previous labs**: Lab 9.1
  - Continue working in your Lab09.1 project
- **Approximate Time**: 20-30 minutes

Modify the Resource

Lab

Tasks to Perform

- Open *pom.xml* for review
  - Look at the Dependency Hierarchy - you'll see the Jackson jars there
- Open `ItemsResource` for editing
  - Change the return value of `findItem()` from `String` to `MusicItem`

```
public MusicItem findItem(…)
```

  - Change the return statement to return the object, not a string

```
return cat.findById(id);
```

  - Upon a request, the returned object can now be marshaled to JSON
- Similarly, change the return value of `getAllItems()` to `Collection<MusicItem>`
  - Change what it returns to be the result of the lookup on the catalog
  - Not the string representation

Review the Ajax JSON Page

Lab

Tasks to Perform

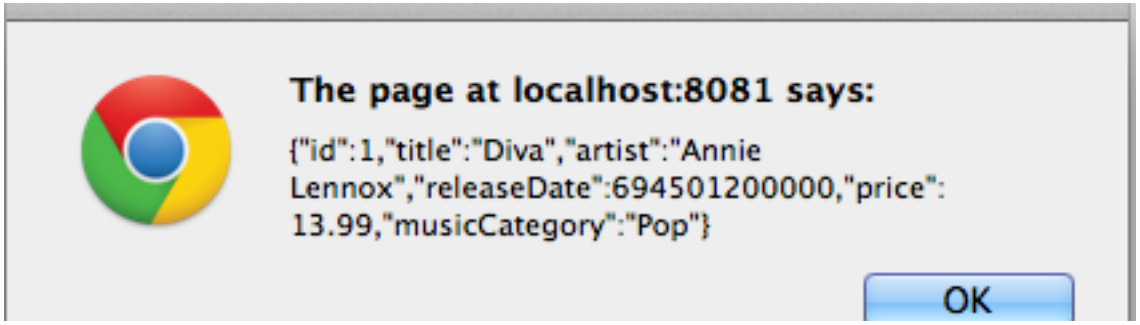
- Open MusicItem for review (**nothing for you** to do here)
  - Note the `@JsonFormat` annotation on the `releaseDate` field
  - The default mapping by Jackson for a `LocalDate` is odd, so we change it via this annotation (1)
- Open *webapp/**views/ajaxJSON.jsp*** for review
  - **No coding needed** in these pages - they are complete
  - The click handlers make ajax calls as before
  - Now they **specify JSON** as the return type
    - You can do a search for json in the file to see this
  - The handlers also do some simple manipulation of the JSON to generate HTML for display
- **Restart** the server, browse to the Web app home page
  - Click on the link to get to the **Ajax JSON page**
  - See details on next page

Test the App

Lab

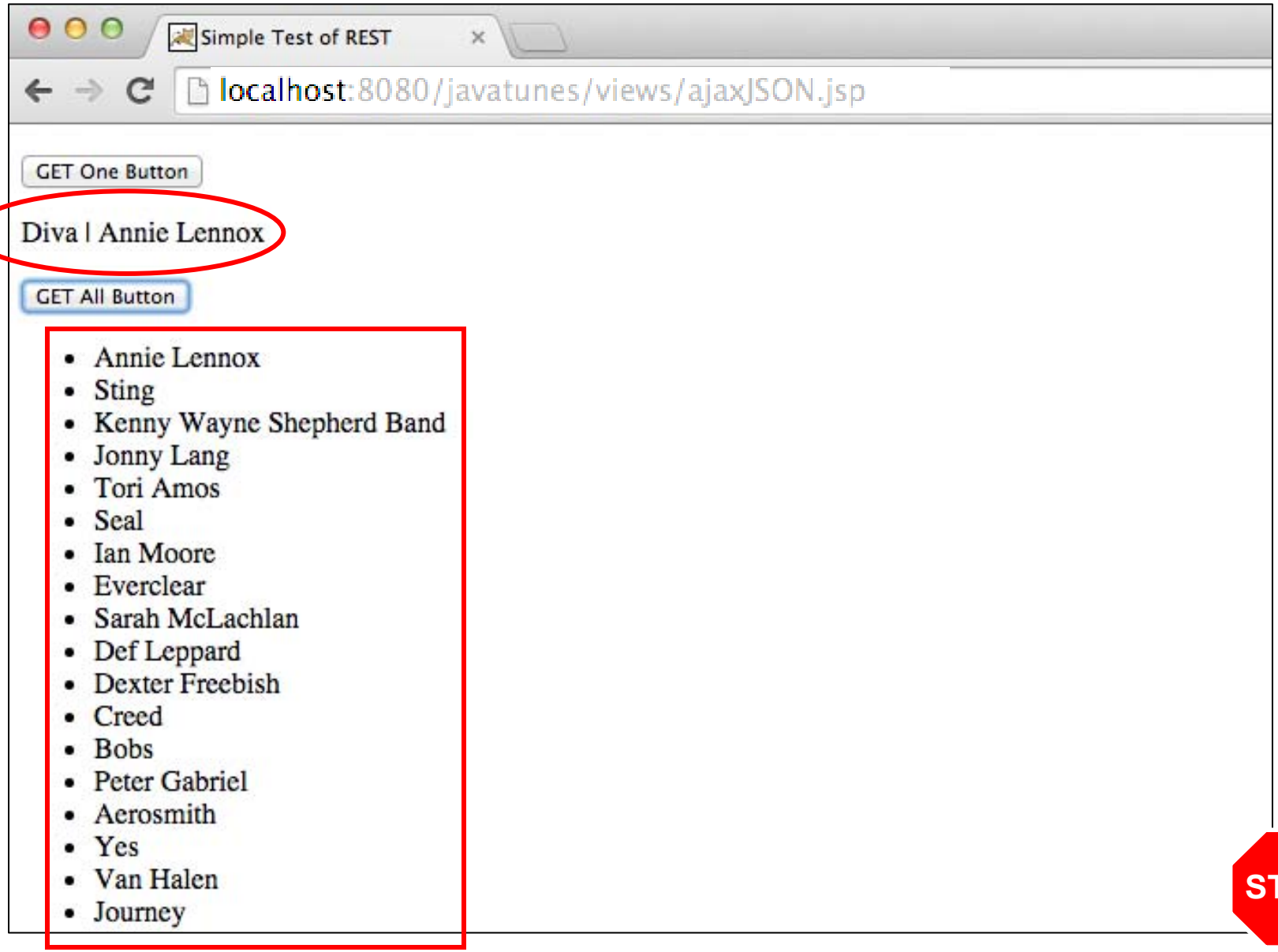
Tasks to Perform

- Try out the buttons - they get REST data via an Ajax call
  - They receive JSON, display it in an alert, process it, then display the processed data
  - The **GET One Button** displays the **title | artist** for the item with `id==1`
  - The **GET ALL Button** displays a **list of artists** of all items
  - See below for JSON in alert box, and see screen shot on next page
- You can see how easy it is to generate and use JSON



Resulting Display

Lab



STOP

LEARNINGPATTERNS

Lab 10.2: An XML Resource

Create a RESTful Resource Returning XML Data

Lab Synopsis

- **Overview**: In this lab, we will work with XML support in Spring MVC/REST
  - We'll create a REST resource that generates XML data
  - We'll access the resource and display the data from an Ajax page
  - We'll use the Jackson XML support, and Jackson equivalents to the JAXB notations
    - They give a little more capability / flexibility than JAXB
- **Builds on previous labs**: none
- **Approximate Time**: 25-35 minutes

Lab Preparation

- The (new) root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab10.2

Tasks to Perform

- **Remove** previous labs from the server, **close** open files/projects (1)
- Import an existing Maven project called **Lab10.2**
  - See earlier lab instructions if you need more detail on this
- Open *pom.xml* and look in the `<dependencies>` element
  - We added the Jackson XML library to our dependencies
  - Notice that there is no version specified
  - Spring Boot specifies a version, and maven pulls it in appropriately

```
<dependency>
   <groupId>com.fasterxml.jackson.dataformat</groupId>
   <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Add Annotations, Finish Resource

Lab

Tasks to Perform

- We'll use Jackson annotations to customize the generated XML
- Open *MusicItem.java* (in package `com.javatunes.domain`)
  - Add the following Jackson annotation

```
@JacksonXmlRootElement(localName="item")
```

  - In `com.fasterxml.jackson.dataformat.xml.annotation`
  - This specifies `item` as the name of the generated XML element (1)
- **Deploy** to the server as usual (restart it), browse to home page
  - Click on the link to get to the Ajax page

Test the App

Lab

Tasks to Perform

- Try out the buttons - they get REST data via an Ajax call
  - They receive XML, display it in an alert, process it, then display the processed data
  - The **GET One Button** displays the **title | artist** for the item with id=1
    - See next slide for XML in alert box
    - The HTML display in the Web page will be the same as with JSON
    - The jQuery code to create it is different - it processes XML not JSON
  - The **GET ALL Button** will display a **list of artists** of all items
    - See slide following next for XML in alert box

GET One Button XML

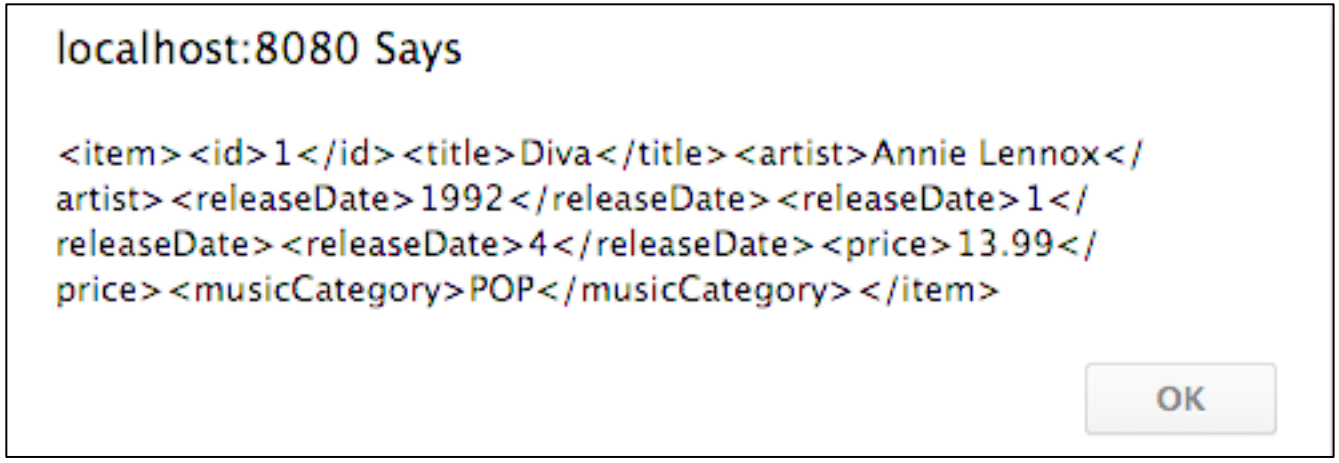
Lab

Tasks to Perform

- The **GET One Button** receives XML as shown at bottom
  - Note how the top level element is `<item>`
  - This is specified by our annotation (1)

```
@JacksonXmlRootElement(localName="item")
```

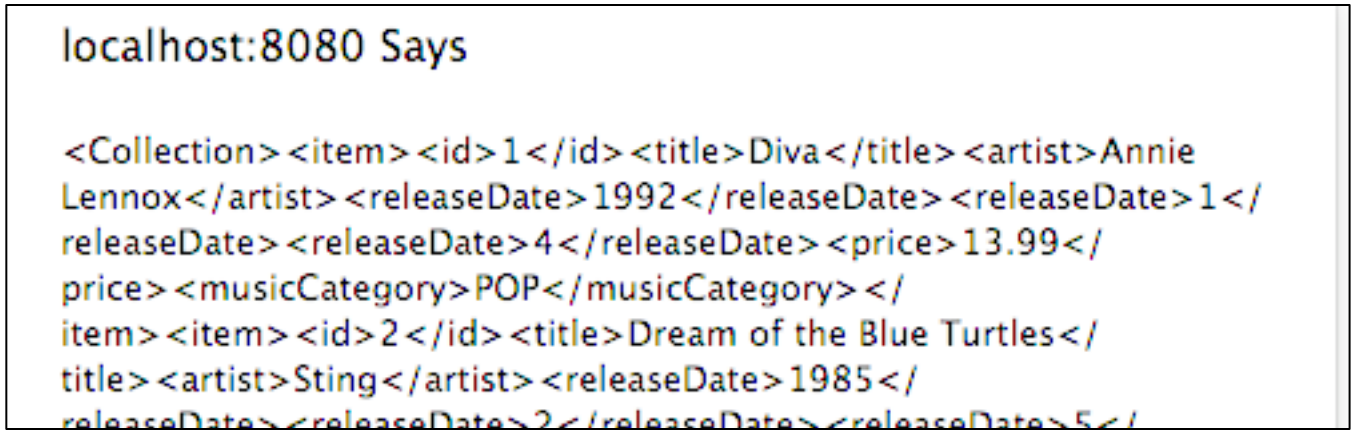
  - The default top level element name is `MusicItem`
  - It's not used because of our annotation
  - Click OK, and you'll see the Web page - which looks the same



GET All Button XML

Lab

- The **GET All Button** receives XML as shown at bottom
  - Note how the top level element is collection of some sort
  - Our resource is returning the collection directly
  - The top-level element name is a default based on the collection type
  - We'd rather have something that makes more sense (e.g. `<items>`)
  - We'll modify it using a wrapper class and Jackson annotations
  - Note that our jQuery code happens to work with this XML also
    - But we'd still like to structure it better

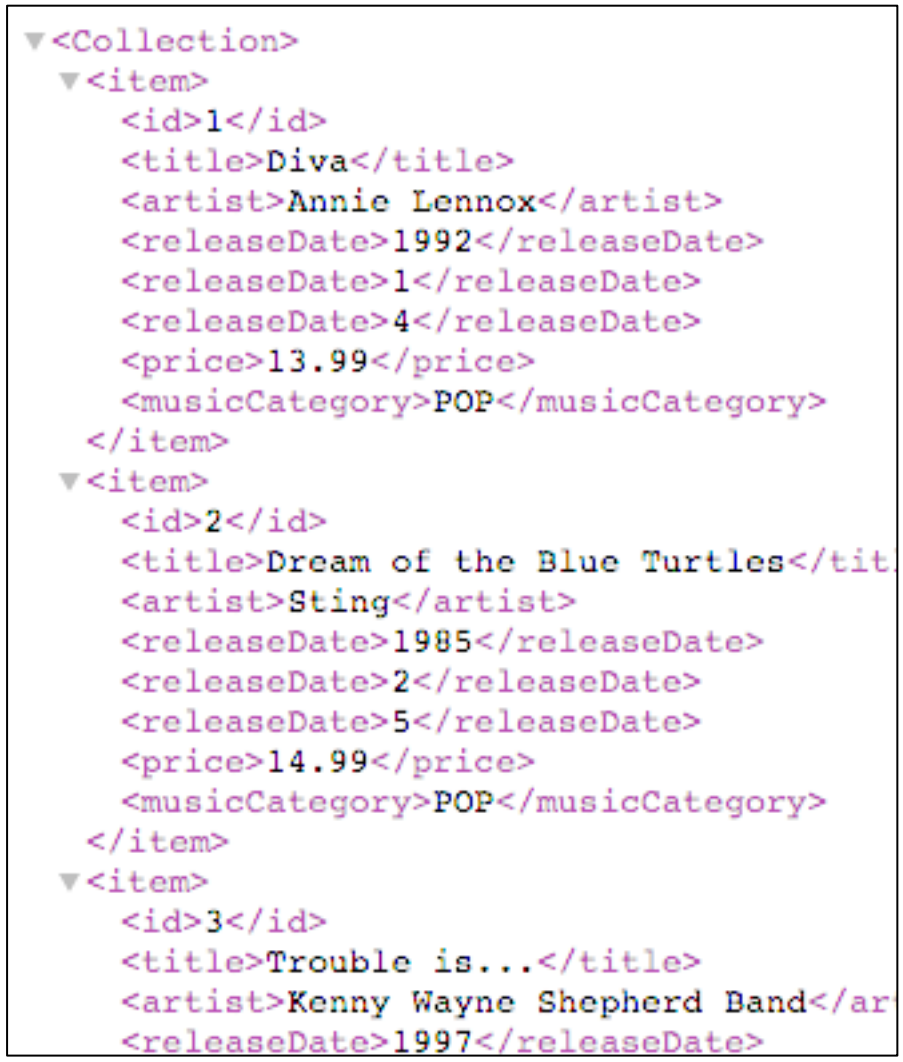


Access the Resources Directly

Lab

Tasks to Perform

- You can access these resources directly in the browser
  - Try going to:
    <host>/javatunes/**rest/items**
  - You'll see the XML, as shown at right
  - Same for one item e.g.:
    <host>/javatunes/**rest/items/2**



[Optional] Control Collection XML

Tasks to Perform

- Open *MusicItemCollectionWrapper.java* for review (in `com.javatunes.domain`)
  - This is a Jackson annotated wrapper we supply - see notes
  - It specifies the XML structure generated by the wrapper class and its contained items
  - **There is nothing you need to do** in this class
- In `ItemsResource`, change the value returned in `getAllItems()`
  - Change the return type to `MusicItemCollectionWrapper`
  - Wrap the collection in an `MusicItemCollectionWrapper` before returning it

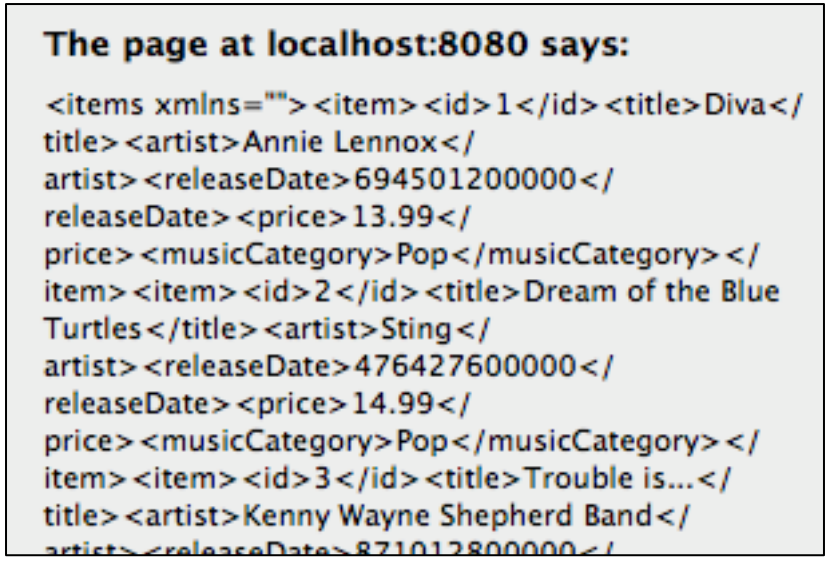
```
return new MusicItemCollectionWrapper(results);
```

- **Restart** the server, and browse to the Ajax page again
  - Click the GET All Button and look at the XML
  - It should have a different structure - with top level element `<items>`

GET All Button XML - Revised

Lab

- The **GET All Button** receives XML as shown at bottom
  - Note how the top level element is `<items>`
  - Our resource is returning the wrapped collection
  - The top-level element name is specified by our annotations
    - If you click OK, the resulting Web page should be the same as before



[Optional] Specify Format Produced

Lab

Tasks to Perform

- Use the `@RequestMapping` elements to specify the data format our resource produces
- In `ItemsResource`, specify that `findItem()` produces only JSON
  - Use the **`produces=`** element of `@RequestMapping`
- **Restart** the server, and browse to the Ajax page again
  - Click the **GET One button**
  - It will no longer work, because the jQuery is specifying it accepts XML, but the RESTful service produces JSON
  - If you want to see the actual return data, you can either:
    - Use tcpmon to examine the response - there are instructions in the next lab on how to use it
    - Use a browser-based JavaScript debugger
- Remove the JSON specification in `ItemsResource.findItem()`, and **restart** the server

[Optional] Examine jQuery code

Lab

Tasks to Perform

- If it's useful to you, you can review the jQuery code that works with the XML
  - For GET One Button, it's directly in the click handler
    - Search for `$('#getOneButton').click`
  - For GET All Button its in a method called `renderArtists`
    - Search for `renderArtists`
- That's it - you've generated XML from a REST resource, as well as consumed it with JavaScript

LEARNINGPATTERNS

Lab 11.1: A Client Using RestTemplate

In this lab, we will access a RESTful resource from a standalone Java client using `RestTemplate`

Lab Synopsis

- **Overview**: In this lab, we will write a client that uses `RestTemplate` to invoke a RESTful service We'll use the services from our previous labs
  - We'll start simply, and in later labs look at more capability

- **Builds on previous labs**: Lab 10.2 for its RESTful resources
  - You MUST have the **Lab10.2 project deployed**, and the **server running**
  - This lab's work is done in a new client project and folder - **Lab11.1**

- **Approximate Time**: 35-45 minutes

Lab Preparation

Tasks to Perform

- Close all open files
- Import an existing Maven Java project called Lab11.1 in the workspace (1)
  - See earlier lab instructions if you need more detail
  - **Do not** switch to a Java Perspective if prompted
- **Note**: We'll be finishing the client in several parts
  - Just finish the part that the instructions you're looking at direct you to

Complete Client Code

Lab

Tasks to Perform

- Open **`RestClient`** for editing (in `com.javatunes.rest.client`)
  - Add the following code - look for the `// TODO` comments
- Look for the definition of `ID_URI`, and finish it by appending a URI Template variable for an item id to the `BASE_URI` variable
  - So we can call our REST service at a URI like *<webapp>/items/**2***
- In **`main()`** , do the following (see the TODO comments)
  - Create an instance of `RestTempate`
  - Use `RestTemplate.getForObject()` to get a `MusicItem`
    - Use `ID_URI` as the URI, `MusicItem.class` as the response type, and a single id value (a string) for the url variables
- **Run** the `RestClient` program and view the results
  - Right click on *RestClient.java*, select **Run As | Java Application**
  - You should see output showing your item in the console window
    - See notes about **viewing the correct console**

Complete Client Code

Lab

Tasks to Perform

- In `main()`'s next code section, use `getForObject()` to get a `MusicItem` as a string
  - Use a response type of `String.class`, and uri and uri variables as in the last call
- **Run** the `RestClient` program again and view the results
  - You should see output showing your item in the console window
  - It should show an XML representation of the item
  - This is because `RestTemplate` is sending the **`Accept`** header

    `Accept: text/plain, application/xml, text/xml, …`
  - Since XML is listed first, it is chosen as the representation
  - We'll see how to change this (if you want) later
- Next in `main()`, use a **map** to pass the URI variable's values
  - The other parameters should be the same as in the first call
- **Run** the program again and view the results

[Optional] Get a Collection

Lab

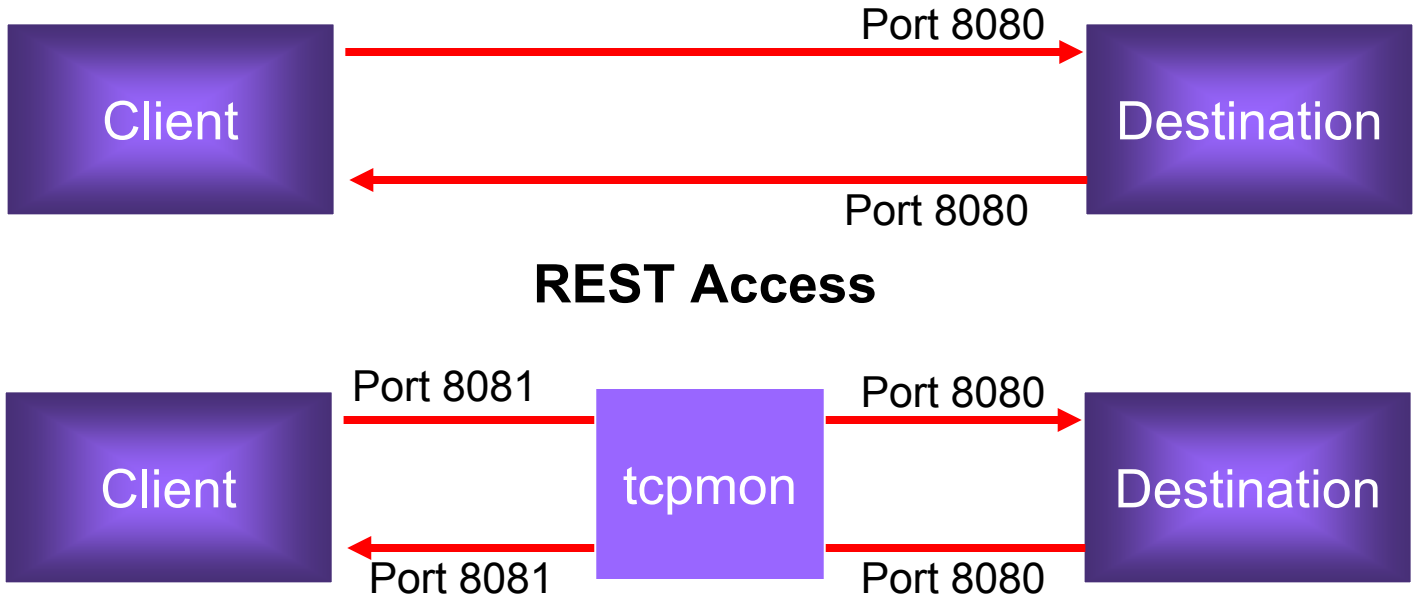
Tasks to Perform

- Back in your service (`ItemsResource` in `com.javatunes.rest`), make sure that `getAllItems()` returns the wrapped list
- Back in the client, in the next code section, use `getForObject()` to get all the items
  - Use a URI of `BASE_URI`, and response type of `MusicItemCollectionWrapper.class`
  - No URL variable values are passed in (`BASE_URI` doesn't specify any URI Template variables)
  - Extract and display the collection that's returned
- **Run** the `RestClient` program again and view the results
  - You should see collection of MusicItem displayed
- You can see how easy it is to use `RestTemplate`
  - We'll go into more of its capabilities later

[Optional] tcpmon Overview

Lab

- tcpmon allows you to intercept TCP/IP messages
  - You simply give it a listen port and a target port, and it will sit in the middle of the requester and responder
  - In the diagram below, the top diagram shows direct access
  - The diagram at bottom shows access through tcpmon



[Optional] Use tcpmon

- You can optionally use tcpmon to view all the traffic between client and server
  - You'll see your REST services at work, and see what requests are being made, and what data is being transferred
  - This is not required, but useful to get a feel of how REST works

Tasks to Perform

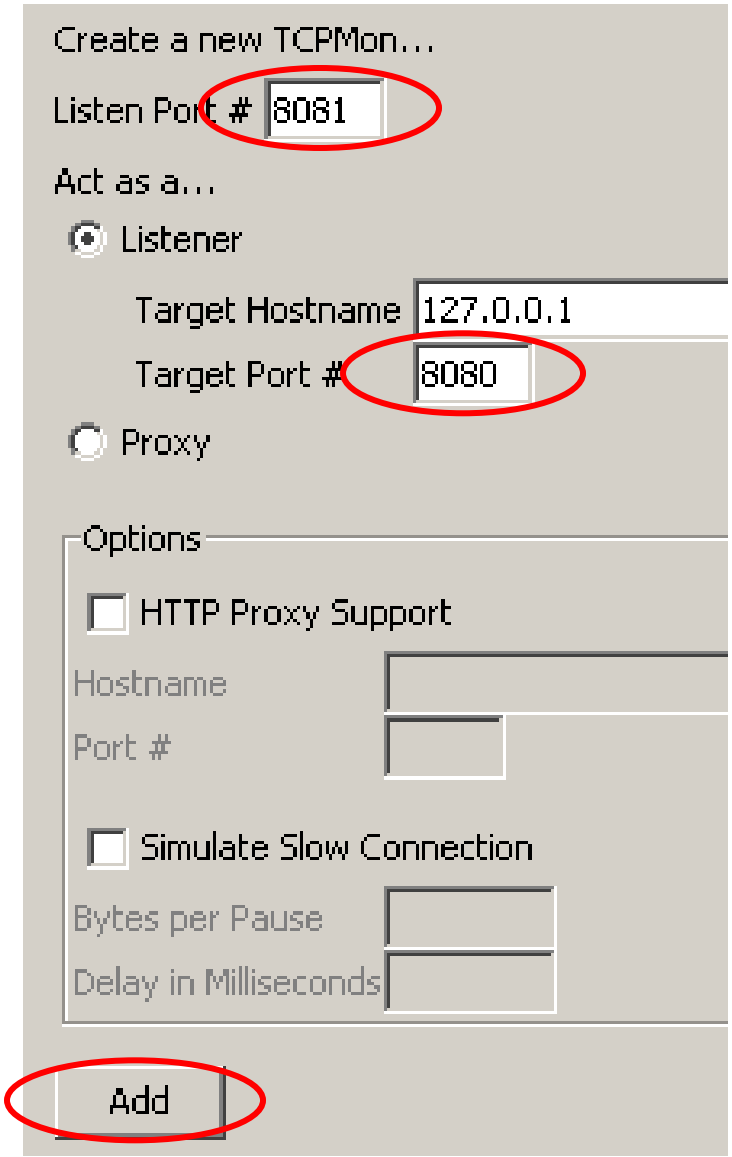
- Go to ***C:\StudentWork\SPRING\tcpmon***, and run **`tcpmon.bat`**
  - You can double click on it - this starts up the tcpmon program
  - It opens in the **Sender** tab
  - Select the **Admin** tab, which we can use to intercept all the traffic between our REST client and the RESTful service
  - Continue as described on the next slide

[Optional] Use tcpmon

Lab

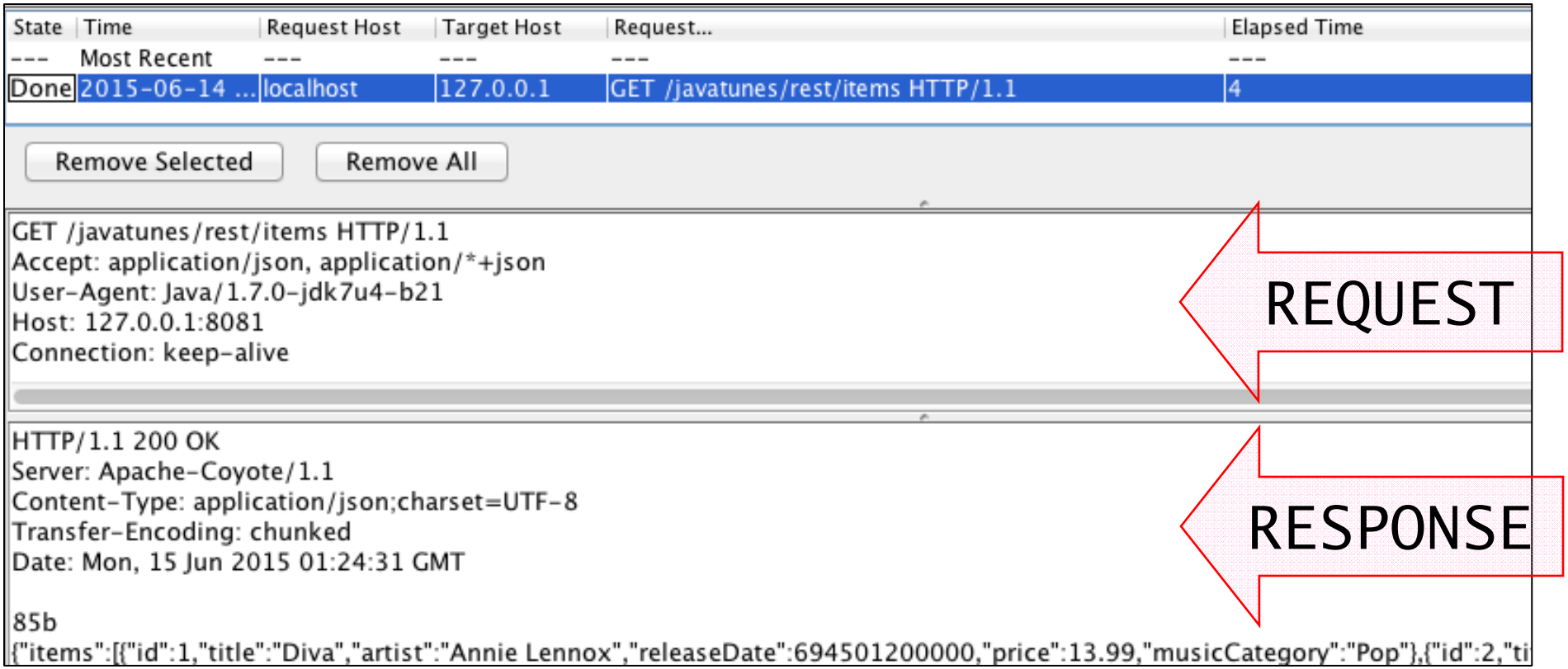
Tasks to Perform

- In the admin tab, fill in the values as follows:
  - Listen Port # **8081**
  - Target Port # **8080**
  - Click the [**Add**] button
- **Click the 8081 tab**, that appears (it shows request/response traffic)
- In your client, modify **BASE_URI** to use port **8081**, instead of 8080
- Run your client
  - tcpmon should show the request/response traffic
  - See next slide for an example



[Optional] The TCPMon Display

Lab



LEARNINGPATTERNS

Lab 11.2: Setting / Accessing Headers

In this lab, we will work with HTTP headers in both the request and response

Lab Synopsis

- **Overview**: In this lab, we'll use additional `RestTemplate` functionality
  - We'll get and set headers when accessing a RESTful service
  - We'll examine the headers from a `getForEntity()` request
  - We'll use `exchange()` to set accept headers
  - We'll examine the headers that are returned
- **Builds on previous labs**: Lab 11.1
  - Continue working in your Lab11.1 project
  - Also depends on Lab10.2 **deployed and running**
- **Approximate Time**: 25-45 minutes (depending on optional parts done)

Use getForEntity()

Lab

Tasks to Perform

- Open *RestClient.java* for editing
  - Comment out all the `getForObject()` invocation and output code
- Add code to do the following
  - Use `RestTemplate.getForEntity()` to get a `MusicItem`
    - Use `ID_URI` as the URI, `MusicItem.class` as the response type, and a single id value of "2" (a string) for the `urlVariables`
    - The return type of the call will be `ResponseEntity<MusicItem>`
    - Access and output the content type and status code from the return value
    - Access and output the body from the return value
- **Run** the `RestClient` program and view the results
  - Observe the content type and status codes of the call
  - The returned item should be a normal music item

[Optional] Set Headers to Accept XML

Tasks to Perform

- Time permitting, do the following
- Create an `HttpHeaders` object
- Create an `ArraryList<MediaType>`
  - Add in `MediaType.APPLICATION_XML` to the list
  - Set the accepts list of the headers to be your list of `MediaType`
- Create an `HttpEntity<MusicItem>` - passing in your headers object to the constructor
- Call `RestTemplate.exchange()`, to GET an item by id
  - Pass in your `HttpEntity` as the request entity in the call
  - The `HttpMethod` should be `HttpMethod.GET`
  - The response type should be `MusicItem.class`, and other parameters as in previous calls
  - Access and output the content type and body from the return value

[Optional] Configure XML Unmarshalling

Tasks to Perform

- If you've used `@JacksonXmlRootElement` or other Jackson XML annotations on your entity classes
  - You must configure `RestTemplate` to handle them
  - Add the code shown at bottom to your program
  - This registers Jackson2's XML support, which can process those annotations properly to convert XML to your Java objects
- **Run** the program - look at the results
  - What is the content type of the return?

```
RestTemplate rt = new RestTemplate();
// Converter is in package org.springframework.http.converter.xml
rt.getMessageConverters().add(
   new MappingJackson2XmlHttpMessageConverter());
```

[Optional] View Raw Data

Tasks to Perform

- If you want to see the data that was returned, do the following
  - Create an `HttpEntity<String>` passing in your same headers object to the constructor
  - Call `exchange()` as before, except use your new request entity, and use `String.class` for the response type
  - Get the body from the return value of `exchange()` and output it
  - **Run** the program, and view the output
- Optionally, change the media type the client accepts back to MediaType.APPLICATION_JSON
  - **Run** your program again - view the JSON returned
- That's it - you've used some of the entity capabilities
  - We'll explore more `RestTemplate` capabilities later

LEARNINGPATTERNS

[Optional] Lab 12.1: Additional REST Operations

Implement additional REST operations on the server side, and test them via a Java client

Lab Synopsis

- **Overview**: In this (optional) lab, we will implement additional REST operations on the server side
  - PUT, DELETE, and POST
    - Most of the code will be given to you - you'll just need to add the code to turn them into RESTful services
  - We'll also write clients for these using `RestTemplate`
- **Builds on previous labs**: None
- **Approximate Time**: 45-60 minutes
  - You can choose which parts to do - they are independent

Lab Preparation

- There are two new folders/projects where you will do your work:
  Server: C:\StudentWork\Spring\workspace**\Lab12.1**
  Client: C:\StudentWork\Spring\workspace**\Lab12.1-Client**

Tasks to Perform

- Remove any projects from the server, close all open files\projects
- Import an **existing Maven project** called **Lab12.1**
  - See earlier lab instructions if you need more detail on importing
- Import an **existing Maven project** called **Lab12.1-Client** in the workspace
  - Do not switch to a Java Perspective if prompted

Finish Controller for DELETE

Lab

Tasks to Perform

- In the **Lab12.1** project (the server) open *ItemsResource.java* for editing, find the `deleteItem()` method, and add the following to it
  - `@RequestMapping` that specifies the correct URI and HTTP method
    - The URI will be something like *<webapp>/items/2*
    - The "2" part will be represented by an URI Template variable, and your controller method will extract this as the id of the item to delete
    - The method is HTTP DELETE
  - `@PathVariable` to bind the URI Template variable to the id parameter
  - Add `@ResponseStatus` to specify NO_CONTENT status
  - `@ResponseBody` is not needed (since we use `@RestController`)
- **Add** the project to the server (see earlier labs if you need details)
  - **Restart** the server

Finish Client for DELETE

Lab

Tasks to Perform

- In the **Lab12.1-Client** project open *RestClient.java* for editing
  - Look for the TODO comments in `main()` that include DELETE
- Add a call to the RESTful DELETE service using `RestTemplate`
  - Use one of the `RestTemplate.delete()` variants
  - Use the simplest method that you can here and in the rest of the lab
    - This is generally the one that takes an Object… argument if you need to pass in URI Template variable values
  - Use a URI with a URI Template variable for the id
- **Run** the client / view the results - the item you deleted shouldn't be present in the collection of all items fetched after the delete
  - **Note**: See notes about running the client more than once

Finish and Test PUT

Lab

Tasks to Perform

- In *ItemsResource.java*, find the `updateItem()` method, and add:
  - `@RequestMapping` that specifies the correct URI and HTTP method
    - The URI will be something like *<webapp>/items/2* (with a URI Template var)
    - The method should be an HTTP PUT
  - `@PathVariable` to bind the URI Template variable to the `id` parameter
  - `@RequestBody` to bind the request body to the `item` parameter
  - **Restart** the server
- In *RestClient.java*, add the following (Look for TODO comment with PUT)
  - Add a call to the RESTful PUT service using `RestTemplate`
  - The URI should include a URI Template var for the id
  - Use the `putId` variable as your id value, and the found (and changed) object we already include in the code as the request object
  - **Run** the client / view the results - the 2nd GET should show new values

Finish Controller for POST

Lab

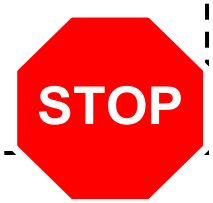
Tasks to Perform

- In *ItemsResource.java*, find the `createItem()` method, and add:
  - `@RequestMapping` that specifies the correct URI and HTTP method
    - The URI will be something like <webapp>/items
    - There is NO URI Template variable in the URI
    - The method should be an HTTP POST
  - `@ResponseStatus` to specify CREATED status
  - `@RequestBody` to bind the request body to the `item` parameter
    - This will be in the method parameter list
  - Within the `createItem()` method, set the **`Location`** header via the `HttpResponse` parameter passed into the method
    - The value will be of the form `/items/idValue`
- **Restart** the server

Finish Client for POST

Tasks to Perform

- In *RestClient.java*, add the following (Look for TODO comment w/POST)
  - Add a call to the RESTful POST service using `RestTemplate`
  - The URI should include a URI Template var for the id
  - Use the already created item (`newItem`) supplied in the code as the request object
  - First use `postForObject()` to create and get the object back
  - Next, use `postForLocation()` to create and get just the location back
    - This second call creates a new object (with a different id) using the same data as in the `postForObject()` call - that's fine for our testing
- **Run** the client / view the results
  - You should see the newly created item output, and then a location URI(with a different id)
- That's it - you've written and used 3 new REST services



LEARNINGPATTERNS

[Optional] Lab 13.1: WebFlux Demo

We illustrate a reactive resource and client written using WebFlux

Lab Synopsis

- **Overview**: In this (optional) lab, we will demonstrate
  - A reactive REST resource that uses Mono and Flux
  - A reactive client (using Spring's WebClient) that consumes the resources in various ways
  - There is NO CODING in this lab
- **Builds on previous labs**: None
- **Approximate Time**: 15-20 minutes

Lab Preparation

- There are two new folders/projects where you will do your work:
  Server: C:\StudentWork\Spring\workspace**\Lab13.1**
  Client: C:\StudentWork\Spring\workspace**\Lab13.1-Client**

Tasks to Perform

- Remove any projects from the server, close all open files\projects
- Import an **existing Maven project** called **Lab13.1**
  - See earlier lab instructions if you need more detail on importing
- Import an **existing Maven project** called **Lab13.1-Client** in the workspace
  - Do not switch to a Java Perspective if prompted

Review the Reactive Resource

Tasks to Perform

- In the **Lab13.1** project, open **ItemsResource**
  - Package `com.javatunes.rest`
  - It has our standard `ItemRepository` injected (an in-memory version)
  - Review **`findItem(Long id)`**
    - Note how it simply wraps the found item in a `Mono`
  - Review **`getAllItems()`**
    - Note how it first wraps the items in a `PausingMusicItemCollection`
    - This is our own collection that simulates pauses in processing between every few elements
    - This helps us demonstrate the "reactive" nature
    - It then wraps this collection in a `Flux`
  - Review any other types of interest - they should be familiar to you

Review the Reactive Client

Lab

Tasks to Perform

- In the **Lab13.1-Client** project, open **`ReactiveWebClient`**
  - Package `com.javatunes.rest.client`
  - Note how it creates a **`WebClient`** on our base REST URI
  - Review the access to `<javatunes>/rest/items/2`
  - Note the creation of a `Mono` from the result
  - Note the subscribing to the `Mono` for processing
  - Review the access to `<javatunes>/rest/items`
  - Note the creation of the `Flux` from the result
  - Note the subscribing to the `Flux`
    - Its processing will be executed for each element
    - It is done asynchronously
  - There is more, we'll review it soon

Run the Server and Client

Lab

Tasks to Perform

- In Package Explorer, **Lab13.1**, Right Click on **BootReactiveDemo**, and select **Run As | Java Application**
  - This is the server - in package `com.javatunes`
- In Package Explorer, **Lab13.1-Client**, Right Click on **ReactiveWebClient**, and select **Run As | Java Application**
  - Package `com.javatunes.rest.client`
  - View the console window for the **client**
    - Make sure you're viewing the correct console (1)
    - Note how you've consumed a `Mono`, and printed out the data
    - Note that execution is paused now, waiting for you to type `Return`
  - View the console window for the **server**
    - Note the DEBUG logging at the end of the server output - it shows the asynchronous handling of the request

View the Flux Processing

Lab

Tasks to Perform

- Review the next part of the client program
  - This accesses **<javatunes>/rest/items**
  - It gets a collection of items as a **`Flux`**
    - Note that there are pauses in the processing on the server side - we've simulated processing time
    - This will result in pauses on the client - but partial results are sent and able to be processed
- Go to the console window for the **client** again
  - **Click in it** to get focus
  - Type **Return**, to move on to the next part of the program
  - Note the output from the `Flux` processing
    - You'll get a couple of items, then a pause, then another couple of items
    - The items are being sent (asynchronously) as they are available

View Filtering

Lab

Tasks to Perform

- Review the last part of the client program (not the commented out part)
  - This also accesses **<javatunes>/rest/items**
  - It adds filtering to the Flux - so you'll only see items with even numbered ids
- View the console window for the **client** again and **click in it**
  - Type **Return**, to move on to this next part of the program
  - Note the output from the `Flux` processing
    - Now you get **even id items** only
    - And still pauses from our server side processing
  - **Type Return** again to exit the client

[Optional] Add a Second Subscriber

Lab

Tasks to Perform

- The client program has code for a second subscription
  - It is commented out - **uncomment** it now
  - Run the client program again, and press Return to go past the Mono processing
  - Now you'll see output from two subscriptions to this data
  - We've put numbers 1 and 2 in front of the output so you can identify processing from the different subscribers
  - These are happening in different threads - which are both asynchronously processing the elements sent

Run a Standard Rest Client

Tasks to Perform

- In the **Lab13.1-Client** project, open **`RestClient`**
  - Package `com.javatunes.rest.client`
  - It should look familiar - it is a standard REST client
- Run this client (Right click, **Run As | Java Application**)
  - You'll see that for the retrieval of all items, the client waits and then all the items come in at one time
  - There is no asynchronous push of individual items

Summary

- We've demonstrated how to write reactive programs with Spring WebFlux
  - We've used parts of the core API
  - We've demonstrated some of the behavior on both the client and server side

- This API makes reactive programming relatively easy to use
  - However, the API is still somewhat complex in syntax
  - There's plenty more to dig into if you want to learn more
  - Look at the WebFlux docs, and the Reactor docs
  - They have a lot of information