##Why Spring?

Spring is very powerful. It's like a huge toolbox with nearly everything you could possibly need. It also has many conventions that force developers to develop cleaner code with well researched design patterns.

Just like a real toolbox, not every tool is needed for every project. A walkthrough of some of the tools that Spring has can be found [here](https://spring.io/guides). After this class, you should feel comfortable enough to finish any of the guides provided by Spring. Familiarize yourself with some of these tools so that when you see a problem, you approach it with the right solution.

##Background of Spring from EJB EJB stands for Enterprise Java Beans and was developed to help with enterprise level java web applications. What was supposed to make development and support of enterprise java applications easier, in practice made them more complex and fragile. The response to this was a set of guidelines an patterns that eventually became the Spring Framework. The Java EE principles that Spring implemented are:

* Servlet API
* WebSocket API
* Concurrency Utilities
* JSON Binding API
* Bean Validation
* JPA
* Dependency Injection
* Common Annotations

We will be utilizing a lot of these as part of Spring Boot, though most of it is well hidden so as a developer you don't have to worry about them.

##What is a Bean? A Spring Bean is just a java object. Almost any type of object that you can instantiate with new or factory pattern can be a bean. The main difference is that beans are created as the application starts up, get added to the context, and is available to be used throughout the lifecycle of the application. This is perfect for things like web servers and database connections that are expensive to create and are built to be highly concurrent.

##What is the Spring context? You'll hear about the Spring Context A LOT while working with the Spring Framework. I like to think of the Spring Context as a bucket of Java objects to be used within your application. When you add a bean to the context, you're adding that object to the bucket of objects that Spring knows about.

Once the context has been created, everything inside that context has access to everything else inside the context. The process of letting classes and beans know about and utilize other beans within the same context is known as wiring. We'll get into some examples of that later.

In the end, beans (Java objects) will get created at startup and be available to be used as they are needed by the other objects in the context of the application. So we'll see a lot fewer new objects get created in our code that shouldn't deal with the instantiation of objects, which leads to cleaner, more concise code.

##XML vs Annotations In the beginning, the context of an app was driven entirely by declarative XML files. It makes sense that this would be the way we'd want to declare our beans, however Java developers didn't like to be forced to use fragile XML for the basis of their code and started creating annotations to provide the same fundtionality.

Now when you see newer Spring apps, you will likely see no XML files at all to drive bean creation. Instead you see only pure Java. In this class we will focus entirely on annotation driven Spring Framework, but if you're interested, there are plenty of online resources to delve into XML driven Spring Framework.

#First App

For our first app we will focus on building an events API, which will store and retrieve events for a system.

We'll use Maven for our dependency management in this class. Many teams use gradle, however for our needs Maven is more than enough for us to use for dependency management. If your team uses gradle, follow their best practices as defined by your team.

##Create A New Maven Project

* To create a new project select Create New Project from the main IntelliJ screen.
* On the next screen click on Maven and click Next.
* In the GroupId field type com.tgt.springboot.bootcamp, springboot for the artifactId, and 0.0.1 for the version. Click Next.
* Type springboot in the Project Name and click Finish.

##Set your compiler

Click Intellij IDEA at the top left and click preferences. Type Java Compiler into the search and select it under Build, Exectution, Deployment -> Compiler -> Java Compiler. Under Target bytecode version select 1.8.

##Maven setup

Below is an example pom file to replace the default pom.xml file. This file tells maven that we want to use the Spring Boot Starter, as defined by the spring framework team, as our base. This will automatically add all required dependencies to run a basic spring boot application.

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.tgt.springboot.bootcamp</groupId>

<artifactId>springboot</artifactId>

<name>springboot</name>

<version>0.0.1</version>

<packaging>jar</packaging>

<description>Spring Boot Bootcamp pom</description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.8.RELEASE</version>

</parent>

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<build>

<finalName>springboot</finalName>

<resources>

<resource>

<directory>src/main/resources</directory>

<filtering>true</filtering>

</resource>

</resources>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<repositories>

<repository>

<snapshots>

<enabled>false</enabled>

</snapshots>

<id>……</id>

<name>……</name>

<url>…….</url>

</repository>

<repository>

<snapshots />

<id>……</id>

<name>……</name>

<url>…….</url>

</repository>

<repository>

<snapshots>

<enabled>false</enabled>

</snapshots>

<id>…..</id>

<name>……</name>

<url>……….</url>

</repository>

<repository>

<snapshots />

<id>………</id>

<name>………</name>

<url>……….</url>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>central</id>

<name>Maven Plugin Repository</name>

<url>http://repo1.maven.org/maven2</url>

<layout>default</layout>

<snapshots>

<enabled>false</enabled>

</snapshots>

<releases>

<updatePolicy>never</updatePolicy>

</releases>

</pluginRepository>

</pluginRepositories>

</project>

Click on Enable Auto-Import in the lower right corner to automatically import all packages when changes happen to your pom file.

##Log setup

There are a lot of options for logging in Spring. A typical pattern is for your code to use the slf4j facade and then utilize either logback or log4j as its implementation. The nice thing about this is that by using the facade, you can easily switch which you want to use without any code changes, just a configuration file change. For our this bootcamp, we'll use logback.

Add a new file to your src/main/resources folder called logback-test.xml. Once you have the file, add the following xml to that file. This will format our logs.

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level="debug">

<appender-ref ref="STDOUT" />

</root>

</configuration>

##Application.java

Once you have a pom.xml file and have downloaded all the required dependencies, all you need to create your first spring boot application is an Application.java file. Right-click on your java folder under main and click New -> Package. Type com.tgt.springboot.bootcamp to create a the required package.

Once the package is created, right click it and select New -> Java Class and type Application.java.

Add the following code to the newly created class. The main function here says to run this Application.java class as a SpringApplication. The @SpringBootApplication tells the spring framework that it needs all the spring boot dependencies, and also to automatically scan for custom beans in the source code. We'll go through examples of that later. For now, just know that you have a fully functioning spring boot application with just this little bit of code.

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

Right click Application.java in your IDE and click Run. You should see the application start up successfully and stay running.

While we do have a working app at this point, the feature set leaves something to be desired. Let's start adding some basic functionality.

##REST Controller

Now that we have a working skeleton of a spring boot app, let's start adding functionality to it. The first part of this will be to add a simple controller to the app. A controller is the entry point to any rest call, which will define the path that needs, the input parameters for the call, and deals with returning the http call. Don't worry too much about the details for now, we'll work through them via an example

First, let's add a new package to our app. Right click the com.tgt.springboot.bootcamp package and click New -> Package. Type controller into the field and now we have a controller package. Right click that package and select New -> Java Class, and enter EventController.

The first thing to do is define the class as a Rest Controller for Spring Boot. You do this with the @RestControllerannotation, like this.

@RestController

public class EventController {

}

The next thing we're going to need define a simple endpoint. We do that with a method and a few annotations. Add the following inside your EventController class:

@RequestMapping(value = "/v1/events/ping", method = RequestMethod.GET)

public ResponseEntity getPing() {

return new ResponseEntity<>("PING!", HttpStatus.OK);

}

##Run the app

Right click on your Application.java and click Run Application. You should see it start up with a message like Started Application in 2.554 seconds (JVM running for 3.071). Once you see that, open up Postman and enter the following url

http://localhost:8080/v1/events/ping

Make sure your call is set to GET and click Send. You should see PING! as the output.

You've successfully created your first API!

##Lab Work (15 min)

###1.

Mapping can be done at the class level. Try adding a @RequestMapping(value = "/v1/events") annotation to the class level (right under @RestController) and re-run your app. The endpoint you hit before should fail. Try instead to hit http://localhost:8080/v1/events/v1/events/ping. There it is! Now update your method request mapping so your original http://localhost:8080/v1/events/ping call works again with the class level mapping.

###2.

Create a new method called getEvents for the /v1/events endpoint. Just return a string for now that says GET /v1/events. It will be very similar to the ping endpoint we've already created.

###3.

Add a class level logger to your controller with the following code:

private static final Logger logger = LoggerFactory.getLogger(EventController.class);

Use that logger to submit info level logs that the methods have been called.

hint: use logger.info...

###4.

Create a new package called dto at the same level as the controller package and a new class called EventDto.java. In that class create two private fields, Integer eventId and String eventMessage. Right click in your class and click Generate.... Inside this option generate your getters/setters. Then do the same for equals, hashcode, and toString. After doing that hit Command + alt + L to automatically format your class. Update your getEvents method to return an empty List of events (hint: use new ArrayList<>()).

#Stereotypes

Stereotypes are a special kind of bean where the class itself defines the bean. You've actually already used a stereotype when you annotated your controller class with @RestController. In this section we will be exploring other kinds of stereotypes.

Many applications follow a 3 tier design of a controller calling a service, where business rules are applied, and with a repository layer to deal with storing and retrieving data. There are a few common stereotypes that come with Spring to annotate these layers, while also providing them as beans to the context.

##Service

One of the most common types of stereotypes is the @Service stereotype. This stereotype does a few things:

1. It automatically creates a single instantiation of the bean in a Singleton pattern and adds that bean to the context.
2. Since the class is part of the context, it has access to all other beans in the context through the concept of "wiring".
3. It provides metadata to the next developer that the intention of this class is to be a service class where business rules should live.

###Creating your first Service

First, add a new service package called service, and add a new class called EventService.java. Annotate the service class with the @Service annotation, so your class looks like this

@Service

public class EventService {

}

We'll come back to this class in a bit.

##Repository

Another stereotype that is used is called a repository. Like the service stereotype, annotating a class with this tells future developers that this class is mainly to be used to save and retrieve data. This DOES NOT mean that it has to connect to a database or any other data store (though most of the time it does), but that the methods will handle that for you. For our example we will be storing our data in memory. There will be examples of database connections at the end of this training.

###Creating your first Repository

First, add a new service package called repository, and add a new class called EventRepository.java. Annotate the repository class with the @Repository annotation, so your class looks like this

@Repository

public class EventRepository {

}

Now add a private field that will store the data in memory. Since we're dealing with events and this is an event store, we will store it as a map, where the key is the eventId.

private Map<Integer, EventDto> eventData = new HashMap<>();

Now let's create some methods to access our data. The first will be to store new data. It is good practice to return the saved value in repositories.

public EventDto save(EventDto newEvent) {

eventData.put(newEvent.getEventId(), newEvent);

return newEvent;

}

And then we'll have a method to retrieve a single event.

public EventDto load(Integer eventId) {

return eventData.get(eventId);

}

##Wiring it all up

So now we have a three tiered application ready to go, but none of the classes are yet talking to each other. The process of doing this is called "wiring" of the beans.

First, we'll wire up the repository to the service. The first step is to add the EventRepository object to the EventService class, since the service depends on the repository.

private EventRepository eventRepository;

This let's the service class know that there is a repository object it is expecting to use. Now, to get it so it actually uses the instantiated bean we created with the @Repository annotation, we need to tell the bean that it expects another bean when it is getting created. Spring encourages constructor based autowiring for this type of scenario. This is a constructor, with all dependent beans as parameters, with an @Autowired annotation.

@Autowired

public EventService(EventRepository eventRepository) {

this.eventRepository = eventRepository;

}

Now the Spring context knows that it needs to create the EventService bean due to the @Service annotation, and is expecting an EventRepository bean to be available to use due during the creation of itself due to the @Autowired on the constructor.

##Connecting the dots Since we have a data class to work with, let's use that object to create some service methods that utilize the repository to get and save data. As you can see, this class is absolutely stateless, which is what we want from our service classes

public EventDto saveEvent(EventDto newEvent) {

return eventRepository.save(newEvent);

}

public EventDto getEvent(Integer eventId) {

return eventRepository.load(eventId);

}

##Lab Work (30 min)

1. In your controller class, add the EventService as a bean and wire it up through constructor base autowiring.
2. Add a getAll method to your repository that returns a list of all your events. (If you're having trouble converting your hashmap to a list, see [this article](http://javaconceptoftheday.com/how-to-convert-hashmap-to-arraylist-in-java/)) The signature should be public List<EventDto> getAll()
3. Add a service method getAllEvents() that calls the new getAll repository method.
4. Update the controller GET /v1/events endpoint to return the getAllEvents()
5. Add logic to the save(EventDto newEvent) method in your repository to generate a random ID if one is not provided in the EventDto. (Hint: use new Random().nextInt() to generate a random integer)

#Controller Tests

Now that we have a working controller, let's create some unit tests.

##Maven Setup

First, we need to be sure we have the right dependencies to run our tests. Add the following dependency to your pom.xml file which will provide the testing framework we'll use.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

##Test Setup

Under src/test/java we're going to add a new package called com.tgt.springboot.bootcamp.controller. This is going to be where our controller tests will live.

Create a new class inside this package called EventControllerTest.java.

Create a new method within this class called pingTest(). Since it is a test we will say the method throws a generic catch-all Exception. If an exception is thrown the test will fail anyway. Below is what the file will look like

public class EventControllerTest {

public void pingTest() throws Exception {

}

}

So far so good, but the class doesn't do anything yet and your IDE is telling you to add some tests to the class. The first thing we need to do is tell Spring how it should be run. To do that we add the @RunWith(SpringRunner.class) annotation at the class level.

The next thing we need to do is tell the spring runner what it should be running. To do that we use the @ContextConfiguration(classes = {EventController.class}) along with the @WebMvcTest(EventController.class) annotation right under it. This says to use that controller class to start building the spring context and that we will be mocking out web calls to that controller. In doing so, the controller becomes a bean that we can use.

Now that we've told the the test context how to build itself, now we need to tell it what to be testing. To do that you add @Test to your method that runs the test. If the method runs without issue, the test passes, if not, it fails.

Your class should now look like this:

@RunWith(SpringRunner.class)

@ContextConfiguration(classes = {EventController.class})

@WebMvcTest(EventController.class)

public class EventControllerTest {

@Test

public void pingTest() throws Exception {

}

}

You can now right click on the test and run it, it should succeed, though it's not testing anything yet.

##Testing your endpoint

To test the endpoint, we're going to use the Spring implementation of Mockito. To do that, we need to add a bit to our controller test class.

First, we'll add the following two fields to the class. The autowired WebApplicationContext is provided by the Spring test context dependency we added earlier.

private MockMvc mockMvc;

@Autowired

private WebApplicationContext webApplicationContext;

And then we will add a method that will set the MockMvc field up.  @Before tells Spring to run this method before running the tests:

@Before

public void setup() {

mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();

}

Now that the objects are set up properly, we can use the mockMvc object to make mock calls to our controller. Add the following code to the pingTest() method:

mockMvc.perform(MockMvcRequestBuilders.get("/v1/events/ping"))

.andExpect(status().isOk())

.andExpect(content().string("PING!"));

This tells the mock framework to make a GET call to /v1/events/ping, expect a 200 response, with the content equal to PING!. If the call fails or gets a different response, the test will fail.

##Lab Work(15 min)

###1.

Try changing the test to hit a different endpoint (like just /ping). Did the test fail? Correct the test back so it succeeds again.

###2.

Create a 2nd test that will test the getEvents() endpoint you've created. Test that the status is ok and that it returns an empty array []

###3.

Try running the entire class of tests now as a single run so that both tests run. Now try running it with code coverage as explained [here](https://www.jetbrains.com/help/idea/running-with-coverage.html). Point the test configuration to the correct package after running. Were you able to see what percentage of code coverage you got?

#Properties

Every good application is property driven. Properties should always be used for things that can change with where the application is deployed. This includes any external dependency such as databases, web APIs, kafka connections; but it can also include internal properties that you may want to change without recompiling the entire application such as timeout configuration, health page info, or number of retries for a process.

##Adding a property file Right click your src/main/resources folder and add a new file called application.properties. Spring will automatically find this file and read these properties on startup. Another option to this file is an application.yml file, which would hold the same properties, just in yml format. For our example we will use a declarative application.properties as it maps easier to kubernetes ConfigMaps.

In this file, add the following:

info.app.description=Spring Boot Bootcamp Event API

info.build.version=@project.parent.version@

info.build.artifactId=@project.artifactId@

spring.application.name=springboot-bootcamp

server.port=9300

The info level properties are basically metadata for Spring Boot. It will drive some things that will be useful later. The @project.parent.version@ and @project.artifactId@ tells maven to populate these properties during the build process, that way they are dynamic to the version you are building.

The only functional property is server.port=9300. This tells Spring Boot that I want the port to be 9300 instead of 8080. Start up your application now and try your previous Postman GET call to http://localhost:8080/v1/events/v1/events/ping. It should fail. Now try http://localhost:9300/v1/events/v1/events/ping. As you can see, the property value drives spring to start the app on a specific port.

There are countless packages that are driven by properties. We will have some examples tomorrow of such projects.

##Custom properties

Adding custom properties is as simples as adding a key value pair to the property file. Here is an example property:

example.property=This is an example property

But now if you want to use the property you have to have spring inject it into your bean. You do that with the @Valueannotation. In your service add the following code:

@Value("${example.property}")

private String exampleProperty;

Now your service has the property available to it.

##Test properties

For test properties you have some options. The first option is to create a common property file under the resources folder in the test package. You then refer to those properties with the following annotation:

@RunWith(SpringRunner.class)

@SpringBootTest(classes = {EventService.class})

@TestPropertySource("classpath:application.properties")

public class EventServiceTest {

@Autowired

private EventService eventService;

@Test

public void getEventTest() {

}

}

The other way to do it is to add the properties directly to the @SpringBootTest annotation like the following:

@RunWith(SpringRunner.class)

@SpringBootTest(classes = {EventService.class},

properties = {"example.property=an example property"})

public class EventServiceTest {

@Autowired

private EventService eventService;

@Test

public void getEventTest() {

}

}

##Property Order

Per the spring documentation, properties will be applied in the following order:

1. Devtools global settings properties on your home directory (~/.spring-boot-devtools.properties when devtools is active).
2. @TestPropertySource annotations on your tests.
3. @SpringBootTest#properties annotation attribute on your tests.
4. Command line arguments.
5. Properties from SPRING\_APPLICATION\_JSON (inline JSON embedded in an environment variable or system property)
6. ServletConfig init parameters.
7. ServletContext init parameters.
8. JNDI attributes from java:comp/env.
9. Java System properties (System.getProperties()).
10. OS environment variables.
11. A RandomValuePropertySource that only has properties in random.\*
12. Profile-specific application properties outside of your packaged jar (application-{profile}.properties and YAML variants)
13. Profile-specific application properties packaged inside your jar (application-{profile}.properties and YAML variants)
14. Application properties outside of your packaged jar (application.properties and YAML variants).
15. Application properties packaged inside your jar (application.properties and YAML variants).
16. @PropertySource annotations on your @Configuration classes.
17. Default properties (specified using SpringApplication.setDefaultProperties).

#Lab Work (10 min)

1. Create a service method that will return a string representation of the exampleProperty
2. In your service test, create a test method that will test the example property method you just created. Don't forget to add your test property to your context in one of the ways described above.

#Service and Repository Tests

This section will discuss unit testing the service and repository we just created. We will go over setup and dealing with mock beans.

##Fixing the Controller tests

If you run your controller tests now you will notice that they will both fail. The error you see is java.lang.IllegalStateException: Failed to load ApplicationContext. This is because you've added a dependency bean to your controller class, but in your test it can't find that bean in the Test Context. In order to get those tests to run we need to give the test context either an EventService bean, or an EventService mock bean. The vast majority of times when this happens you want to use a mockbean for this. The reason being that these unit tests are testing the controller, so we don't want real code being run that isn't the controller code.

To add a mockbean to this class add the following field to the test:

@MockBean

private EventService eventServiceMock;

If you re-run your tests, they should now pass. If you want to make your tests as verbose as possible, you could update them to look like the following. This would be the way I'd write the tests as it tells the test exactly how to behave, and precisely what to expect.

@Test

public void pingTest() throws Exception {

mockMvc.perform(MockMvcRequestBuilders.get("/v1/events/ping"))

.andExpect(status().isOk())

.andExpect(content().string("PING!"));

}

@Test

public void getEventsTest() throws Exception {

when(eventServiceMock.getAllEvents()).thenReturn(Arrays.asList(new EventDto()));

mockMvc.perform(MockMvcRequestBuilders.get("/v1/events/"))

.andExpect(status().isOk())

.andExpect(content().string("[{\"eventId\":null,\"eventMessage\":null}]"));

}

@Test(expected = Exception.class)

public void getEventsExceptionTest() throws Exception {

when(eventServiceMock.getAllEvents()).thenThrow(new RuntimeException("message"));

mockMvc.perform(MockMvcRequestBuilders.get("/v1/events/"))

.andExpect(status().isOk())

.andExpect(content().string("[]"));

}

##Service Unit Tests

In our test packages, add a service package and a new class called EventServiceTest.java (right click on java under tests and add the package com.tgt.springboot.bootcamp.service). In that class add the same class annotations as your controller test and add a test method called getEventTest(). We will also autowire our EventService bean so we can use it to test. Your class should look like this:

@RunWith(SpringRunner.class)

@SpringBootTest(classes = {EventService.class})

public class EventServiceTest {

@Autowired

private EventService eventService;

@Test

public void getEventTest() {

}

}

In this test you'll notice we're using the @SpringBootTest annotation. This is telling our test to spin up a spring boot test context and to look at the included classes for beans. Since we're testing the EventService, we tell it

Try running the test. You should see it fail due to not being able to build the application context. This means that the test context, while adding the EventService bean, couldn't find a dependent bean. We know the EventService needs an EventRepository bean in order to run, so we need to add it to our test context. We do that through a concept of mocking out those dependent beans. Below is an example of adding a mock bean in spring:

@RunWith(SpringRunner.class)

@SpringBootTest(classes = {EventService.class})

public class EventServiceTest {

@Autowired

private EventService eventService;

@MockBean

private EventRepository eventRepositoryMock;

@Test

public void getEventTest() {

}

}

A mock bean is exactly what it sounds like, it's an object that stands in place of the real object, but provides none of the functionality. If you call any of the methods in the mock object, it will return null.

Let's add some test code to our method now.

@Test

public void getEventTest() {

EventDto outputEvent = eventService.getEvent(1);

assertNotNull(outputEvent);

}

What is this test doing? It's utilizing the autowired eventService to get an event, then we are validating that the output of that method call returns us something as a basic test. Run the test.

Why is it failing? It's because inside the getEvent method call we're calling the repository to get the data. As stated above, all method calls to mock objects will return null, which is what is happening here. The assertNotNull static method fails the check because it is null.

Let's add some code to tell the mock service what to do when it gets called. Looking at the method we see that the service method calls eventRepository.load(eventId). Below is how we tell the mock object to behave when it gets called.

Mockito.when(eventRepositoryMock.load(Matchers.eq(1))).thenReturn(new EventDto());

This is telling the mock object that when its load method is called with an argument of 1, to return a new EventDto object. Your test method should now look like this:

@Test

public void getEventTest() {

Mockito.when(eventRepositoryMock.load(Matchers.eq(1))).thenReturn(new EventDto());

EventDto outputEvent = eventService.getEvent(1);

assertNotNull(outputEvent);

}

Run the test and it should now succeed.

##Mocking Patterns, Matchers, and Return Values

There are many different ways to tell how the mocks should react. This section will describe a few of the most common.

###Mocking Exceptions

Mocking exceptions is similar to mocking real values, you just use the thenThrow method instead of the thenReturnmethod.

Mockito.when(eventRepositoryMock.load(Matchers.eq(1))).thenThrow(new RuntimeException());

You can then annotate your test method to expect it with @Test(expected = RuntimeException.class). Add the following test to your class to see it work in action.

@Test(expected = RuntimeException.class)

public void getEventExceptionTest() {

Mockito.when(eventRepositoryMock.load(Matchers.eq(1))).thenThrow(new RuntimeException());

eventService.getEvent(1);

}

###Capturing values You can capture inputs to a mocked object using an ArgumentCaptor. The first step of using an argument captor is creating the object to store the captured value.

ArgumentCaptor<Integer> eventIdCaptor = ArgumentCaptor.forClass(Integer.class);

Then we use a verify statement on the mock object *after* the method has been used.

Mockito.verify(eventRepositoryMock).load(eventIdCaptor.capture());

Once the value has been captured, we can retrieve it and test that it was the value we were expecting.

assertEquals(1, eventIdCaptor.getValue());

Your test should now look like this:

@Test

public void getEventTest() {

ArgumentCaptor<Integer> eventIdCaptor = ArgumentCaptor.forClass(Integer.class);

Mockito.when(eventRepositoryMock.load(Matchers.eq(1))).thenReturn(new EventDto());

EventDto outputEvent = eventService.getEvent(1);

assertNotNull(outputEvent);

Mockito.verify(eventRepositoryMock).load(eventIdCaptor.capture());

assertEquals(1, eventIdCaptor.getValue());

}

Note: If your IDE screams about the assertEquals, try assertEquals((Integer) 1, eventIdCaptor.getValue()); to tell it that it's an integer.

###Verifying the number of times a mock method was called

One useful test is to be sure a method was called (or wasn't called) a certain number of times. To to this you just need to update the verify statement like so:

Mockito.verify(eventRepositoryMock, Mockito.times(1)).load(eventIdCaptor.capture());

###Mocking void methods

If the method you want to mock is a void method, you don't actually have to do anything, since a null return value is expected on void methods. It is good practice to use verify statements on the arguments passed in, and there is a bit of a syntax change for mocking exceptions for void methods. To do that, you need to use the following syntax. As you can see, this can be used for non void methods as well, this is just so you have the syntax when needed in the future.

Mockito.doThrow(new RuntimeException).when(eventRepositoryMock).load(Matchers.eq(1));

###Matchers So far we've only used the Matchers.eq method, however there are many kinds of matchers.  eq says that the object must be equal. You can also use Matchers.any(<object-type>.class), which will just verify that the object is the correct type. There is also anyString() or just any(). The API spec for the Matchers can be found [here](https://static.javadoc.io/org.mockito/mockito-core/2.7.0/org/mockito/ArgumentMatchers.html)

##Lab Work (20 min)

1. Add a test to your class to test the saveEvent method. Be sure add a happy path and an exception test. Try using Mockito.when(eventRepositoryMock.save(Matchers.any())).then(returnsFirstArg()); to return the saved object passed into the repository as its return value.
2. Add a repository package and test class. There are no mock objects here, so have your test save the object, then search for the object.

#Creating Data

In this section you'll learn about data transfer objects (DTOs), how to create data through an API call, and validations.

##DTO Objects

Data Transfer Objects, or DTOS, are how we translate data into java objects. That data can be XML, JSON, database data, encrypted data, or really anything. For our purpose we will focus on translating to and from JSON.

###JSON

JSON is just a way to present data in a human readable way. At its base it is a key/value system with the ability to have complex values through curly brackets, indicating an object, square brackets indicating a list, as well as your standard numbers and strings.

For our example we will utilize our EventDto as the base. In that class we have Integer eventId and String eventMessage as fields. The JSON example of this object could be:

{

"event\_id": 1234,

"event\_message": "This is an event message"

}

###Java Annotations

In order for the translation to/from java, we need to tell the object how it should serialize and deserialize the object. We do that through some simple annotations.

public class EventDto {

@JsonProperty("event\_id")

private Integer eventId;

@JsonProperty("event\_message")

private String eventMessage;

//getters/setters/equals/hashcode not shown below

}

That's all that will be needed for Spring to know that this object will be displayed and read in JSON.

##POST Controller

Now that we have an object that can be serialized and deserialized, we should add a method to our REST controller that can save new data. To do that we will create a similar method in our EventController as the getEvents method.

@RequestMapping(method = RequestMethod.POST)

public ResponseEntity saveEvent(@RequestBody EventDto event) {

return new ResponseEntity<>(eventService.saveEvent(event), HttpStatus.CREATED);

}

The differences between this method and our GET methods are:

1. The @RequestMapping annotation now has the method as a POST
2. The method takes in a body via the @RequestBody annotation, which will convert the json to the object for you.
3. We will use the output of the saveEvent service method as the body of the response
4. The status of the response will be 202 - Created, instead of 200 ok

Start up your application and use postman to send a POST to your endpoint with the following data. Be sure to add the Content-Type: application/json header along with it:

{

"event\_message": "this is a new message"

}

You should see it return back with an event id. Now hit your getEvents endpoint, which should see this new data.

##JSR-303 Validations

If you try to send a blank json message like {} your post call will succeed and save the null values. We don't want this, so we will use Spring to do some validations for us. This is already built into the modules we're using, so using a simple annotation we can ensure that the message isn't empty.

public class EventDto {

@JsonProperty("event\_id")

private Integer eventId;

@JsonProperty("event\_message")

@NotEmpty

private String eventMessage;

//getters/setters/equals/hashcode not shown below

}

The @NotEmpty annotation tells spring that the String shouldn't be null or be an empty String. There are many, may built in validations with @NotNull, @Min, @Max, @Size, and @Pattern being very common ones. The full list can be found [here](https://docs.jboss.org/hibernate/beanvalidation/spec/2.0/api/).

This is only the first step, however. You'll now need to tell the controller that it needs to perform the validation. To do that all you'll need to do is add the @Valid annotation to your method.

@RequestMapping(method = RequestMethod.POST)

public ResponseEntity saveEvent(@Valid @RequestBody EventDto event) {

return new ResponseEntity<>(eventService.saveEvent(event), HttpStatus.CREATED);

}

Now if you try to submit an empty JSON the controller will reject it saying that the eventMessage cannot be empty.

##Lab Work (45 min)

1. Run your controller unit tests. Notice that they are now all failing. Update the expected result to be the newly defined json (Hint: event\_message vs eventMessage)
2. Write a unit test that tests the new endpoint with a POST call instead of a get. You'll need to use the .content(jsonString) and .contentType(MediaType.APPLICATION\_JSON) so it looks like mockMvc.perform(MockMvcRequestBuilders.post("/v1/events").content(jsonString).contentType(MediaType.APPLICATION\_JSON)). You'll also need to mock out the eventService so it returns an event. Ensure that you get a created message back and that you get the correct EventDto object.
3. Write a unit test that tries to POST an empty JSON object. You should expect a 400 response back from this test.
4. Update the saveEvent controller method so that if the service call fails, it returns a 500 with a message of

{

"message": "There was an error accessing the service"

}

1. Write a unit test that throws an exception when the eventService is called. You should expect a SERVICE\_UNAVAILABLE (500) message back with the above JSON.
2. Write a new GET getEvent method that takes in an @PathVariable Integer eventId and returns the event associated with that id. The path for is method will be /{eventId}.
3. Write a unit test for the new getEvent endpoint.

#Configurations

Configuration classes in Spring are a way for the developer to add beans through object instantiation. A typical use case for this is when you need an object for your context, but the code for it was written by a third party or is in another application. Another use case would be to organize third party application level annotations to keep your Application.java class clean.

##The Config Class

Add a new package to your project called config and a new class called Config.java. This will be the basis for our configuration class. To tell spring that it should look into this class for configurations, you just add @Configuration to the class. Because we're using @SpringBootApplication in our Application.java class, spring will automatically scan through all our files and look for the @Configuration annotation.

Now that we have a configuration class, let's create a bean. Just as an example we will use an ObjectMapper, which is a pretty common bean to add to your context. To add this to the context we use the @Bean annotation with a method that defines how to create the bean. Below is an example of what that method will look like:

@Bean

public ObjectMapper mapper() {

return new ObjectMapper();

}

That's it! Now we have this bean in our context and can utilize it across our application.

##Wiring Up Your Beans

In your service class, let's add a new ObjectMapper field and autowire it. First, define the field:

private ObjectMapper mapper;

Then update your autowired constructor to populate the field with the bean we find in the context.

@Autowired

public EventService(EventRepository eventRepository, ObjectMapper mapper) {

this.eventRepository = eventRepository;

this.mapper = mapper;

}

This is just as an example. We won't actually be utilizing this bean today, but be aware that the same mapper bean is available to all other beans within the context. Many times you don't need a configuration class as most packages are developed to automatically create beans based on property configuration or are custom to your project. But in some cases you do need some custom beans that are not a part of any package, and this is how you do it.

##Annotation Driven Configurations Some configurations are driven by annotations. Kafka integrations, asynchronous processing, caching, and others can be configured by adding annotations to a configuration class. These configuration annotations will usually add beans to your context automatically, while also telling spring that you're expecting processing to happen differently than vanilla spring.

Here is an example of an annotation that will enable asynchronous processing for methods define as @Async. Again, we won't be utilizing this today, but it is something to be aware of as you go forward in your Spring journey.

@EnableAsync

@Configuration

public class Config {

//snipped for example purposes

}

#Spring Boot Actuator

Spring boot actuator is a *very* powerful tool to automatically create health endpoints for your Spring Boot application. Many packages will automatically configure health endpoints and add them to the actuator, such as Spring Cloud Stream (kafka streams), Spring JPA, and Kubernetes packages.

##Adding the dependency

You can add the dependency directly through Spring, however, we will use a 3rd party dependency that will also give us the added ability to connect to Spring Boot Admin, which we will discuss later. Add the following to your pom.xml file:

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-starter-client</artifactId>

<version>${spring.boot.admin.version}</version>

</dependency>

Then add the following property:

<properties>

<spring.boot.admin.version>1.5.4</spring.boot.admin.version>

</properties>

##Viewing the endpoints

Restart your application after adding the above and navigate to http://localhost:9300/health. You should see some JSON indicating it's up. Actuator does protect the data, however, so to unprotect it you will need to add the following property to your application.properties file:

management.security.enabled=false

Restart your app and hit your health endpoint again. You should see all the data surrounding the health of your app. One of the benefits of this is that if any part of your app shows up as DOWN, the API call will give a 500 error. This is perfect for determining pod health inside kubernetes, which we will get to later.

##Adding custom health attributes

There are some scenarios where you would like to add custom health metrics. Actuator makes this very easy with the HealthIndicator interface. We'll add a simple health indicator to our app as an example. First, create a health package like we have done before. Inside that package add a new class called EventHealthIndicator.java. Add the following code to that class:

@Component

public class EventHealthIndicator implements HealthIndicator {

@Override

public Health health() {

return Health.up().withDetail("EventHealth", "EventHealth is ok!").build();

}

}

The above is just an example, but the main point is that you'll implement a health() method that will tell you the health of your the attribute you are checking. Restart your app and hit the health endpoint and you should see it return ok.

##Spring boot admin

We won't go into too much detail here, but here is an example of the Purchasing Team's instance of Spring Boot Admin in Prod:

This takes advantage of the data provided in our actuator to see things like the version deployed, health, actuator metrics, environment variables, JMX information, and thread information. If you are interested in learning more about this, feel free to contact me on how to set it up in your kubernetes environment.

##Lab Work (10 min)

1. Update your health indicator from Health.up() to Health.down(). Restart your app and hit your health endpoint. What differences were there?
2. Revert your changes from above so it stays up.

#Prometheus

Prometheus is used for metrics collection. Spring Boot makes metrics collections very easy and allows for all kinds of metrics to be captured and created.

##Setup

First we need to add the dependencies to our pom.xml file. We will be using micrometer since Spring Boot 2.0 will come with micrometer out of the box.

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-spring-legacy</artifactId>

<version>${micrometer.version}</version>

</dependency>

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-registry-prometheus</artifactId>

<version>${micrometer.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-hateoas</artifactId>

<version>${spring.boot.version}</version>

</dependency>

And then add the following new property:

<micrometer.version>0.12.0.RELEASE</micrometer.version>

Restart your app and go to http://localhost:9300/prometheus, you should now see a page that is in the format the Prometheus will be able to understand. Run some gets or pings, then hit your prometheus endpoint again. You should see some new datapoints with how long those calls took and the number of calls. This data can be used for graphing our metrics later.

##Custom measurements

All API calls inbound and outbound will be captured with just the above, but if you want a custom metric, it's very easy to do so. Say we want to capture how long it takes to run our getEvents service call. In our EventService class add the following fields:

private MessageMetric messageMetric;

private MeterRegistry metricsRegistry;

private Timer timer;

Then in your @Autowired constructor, wire up the beans available and set up your timer:

@Autowired

public EventService(EventRepository eventRepository, ObjectMapper mapper, MessageMetric messageMetric, MeterRegistry metricsRegistry) {

this.eventRepository = eventRepository;

this.mapper = mapper;

this.messageMetric = messageMetric;

this.metricsRegistry = metricsRegistry;

timer = Timer.builder("events.timer").register(metricsRegistry);

}

Now that your timer is set up, inside your getEvents update it so it looks like:

inboundTimer.record(() -> {

//all the code current inside this method should go here

});

Now you'll get a custom timed measurement of the method.

##Kubernetes Scraping and visualize.abc.com

For future reference, if you're deployed to kubernetes, all you need to do to get these measurements into visualize.abc.com is have a service.yml file that looks like the below:

apiVersion: v1

kind: Service

metadata:

labels:

name: springboot-bootcamp

name: springboot-bootcamp

annotations:

prometheus.io/scrape: 'true'

prometheus.io/path: /prometheus

prometheus.io/port: '9300'

spec:

ports:

# The port that this service should serve on.

- port: 9300

# Label keys and values that must match in order to receive traffic for this service.

selector:

app: springboot-bootcamp

This will tell the kubernets scraper to hit this endpoint and send the data to visualize.

#Kubernetes and Fabric8

Most of our deployments will be to the cloud in Kubernetes. This section will walk you through how to set up your app to be kubernetes ready and give you the flexibility you need to run the same image across multiple environments.

##Setup

As with other packages, we will need to add the necessary dependencies to our application:

<dependency>

<groupId>org.jolokia</groupId>

<artifactId>jolokia-core</artifactId>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

</dependency>

<dependency>

<groupId>io.fabric8</groupId>

<artifactId>spring-cloud-starter-kubernetes</artifactId>

<version>${fabric8.k8.version}</version>

</dependency>

And as with our other dependecies, we need to add the following property:

<fabric8.k8.version>0.1.6</fabric8.k8.version>

##Spring Actuator

Restart your application and hit your health endpoint as we have before. You should now see a kubernetes section, which means the package was successfully added to your application. It should say that it is outside a kubernetes environment, which is what we want.

##Integrating with K8

Now that we have the package in our application, it will interact with the kubernetes environment it is deployed on. What this means is that it will add kubernetes information to the health endpoint, but also allow it to read what are called ConfigMaps and Secrets in kubernetes. The only requirement to get this to work is that your kubernetes Deployment, Service, Secrets, and ConfigMaps *must* all be the same name.

So if our application is deployed as springboot-bootcamp, our service name, secrets name, and configmap all must be named springboot-bootcamp as well. Here is an example of some configurations of the Purchase Orders v2 API for reference:

As you can see, all the name portions are the same and the configmap.yml matches nearly 1:1 to our application.properties file. You can also see some of this in action in the spring boot admin page here:

#Other Topics

This is a page that will gloss over a few topics that should be used as a future reference.

##Singleton vs Prototype Beans

Beans by default are of the Singleton pattern. That means that when you use a bean, you get the same exact bean across every part of the application. These are perfect for stateless classes. Another type of bean is the prototype bean. A prototype says that each time you want to use a bean you actually do create a brand new version of it. This scope is good for classes that contain a state. To do so you simply annotate the bean with @Scope("prototype"). For more examples on other types of scopes, [this](http://www.baeldung.com/spring-bean-scopes) is a good article.

##Spring Kafka

Kafka is big. We use it to pass messages across systems with well defined contracts as to what those messages are. Message queues are great for the Pub/Sub pattern, where you have one or many publishers of the message, and one or many subscribers. How these get linked together is through what Kafka calls 'topics'. The publisher publishes to a kafka topic, it is placed on one of the partitions of the topic (default is 3 partitions per topic) and the subscriber will read off the topic. If the producer produces to the topic with a key, that key will always be published to a single partition, otherwise the message is produced in a round robin fashion.

The consumer is tied to a consumer group and that consumer has an offset tracker for each of the partitions in the topic. Once the message has been acknowledged, the offset is moved to the next message. If the offset is not moved, the message will automatically be retried until it is acknowledged.

You could have another entire training on Spring Kafka over multiple days (and may do so in the future), but for future reference, the below links have Kafka details, Spring Kafka reference docs, and two example apps of Spring Kafka in action.

##Test Suites

Test Suites are really useful to run all your tests in one fell swoop. To add one to your project right click on your com.tgt.springboot.bootcamp package under test and add a new java class called EventsTestSuite. Add the following annotations to tell it to run all the tests you define here:

@RunWith(Suite.class)

@Suite.SuiteClasses({

/\*\*\*\*\*\* CONTROLLER TESTS \*\*\*\*\*\*\*\*\*\*/

EventControllerTest.class,

/\*\*\*\*\*\* SERVICE TESTS \*\*\*\*\*\*\*\*\*\*/

EventServiceTest.class,

/\*\*\*\*\*\*Repository TESTS\*\*\*\*\*\*\*\*\*\*/

EventRepositoryTest.class,

})

public class EventTestSuite {

}

If you run the tests in this class, it will now run all of your tests at once! It's really nice to know if the code you're writing will fail at build before you actually build. ##Spring Security

Spring security can be leveraged to handle the security of your app.

##Aspect Oriented Programming

Aspects can be used as an intercept to methods. The premise is that you can take a method, get its arguments before the method is run, do some things to those arguments, then choose to run the method (or not) and also intercept the output of that as well before returning to the caller of that method

##Future of Spring Boot with Spring Boot 2.0