Introduction to Spring Framework Workshop

Exercise Guide

Table of Contents

Using this Guide

This guide assumes the following:

- You have completed all prerequisites for the course
- You understand how to load projects into your IDE
- You know how to run applications and application servers in your IDE

Conventions Used in the Guide

- When referencing the project files, ** is used to indicate the folder created when you executed the git clone command to download this guide and code materials.
- Once a project is referenced, it will be used for the entire exercise
- All files and packages, unless otherwise specified are the /src/main/java directory

Exercise 1

This exercise will take the information you have learned about the Application Context and Lifecycle and configure an application context for the basic structure of this application. As we haven't yet dug into the details, the key here is to build the configuration a little blind and we will build off of it as we go forward.

Working with **/spring-intro-workshop/code/exercise_1/exercise project

Step 1: Manage Dependencies

Spring provides a Platform BOM to manage dependencies. While you don't have to use it, doing so provides a known set of versions of spring projects and common components to leverage. So for the purposes of this exercise, we will leverage the platform.

In your pom.xml

Create a dependencyManagement section by adding the dependencyManagement xml node after the project version.

```
<dependencyManagement>

</dependencyManagement>
```

Now to add the actual platform POM, note this example is the latest at the time of writing this guide, but please use the latest and greatest platform. Note that leveraging the platform will manage compatible versions for us.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.spring.platform</groupId>
            <artifactId>platform-bom</artifactId>
            <version>Brussels-SR4</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Now we need to add in some Spring Dependancies to our project. With the platform bom you don't have to provide versions for the dependancies that are included as part of the platform. Create a dependencies section after the dependencyManagement section

```
<dependencies>

</dependencies>
```

Create dependency entries for the following. Note the compile scope doesn't need to be actually specified since it is the default.

Group	Artifact	Version	Scope
org.springframework	spring-core	n/a	compile
org.springframework	spring-beans	n/a	compile
org.springframework	spring-context	n/a	compile
org.apache.commons	commons-lang3	n/a	compile
commons-logging	commons-logging	n/a	compile
junit	junit	n/a	test

If your IDE doesn't automatically execute Maven when the POM changes, be sure to execute the dependency goal for maven such as mvn dependency:resolve.

Step 2: Create Java Config

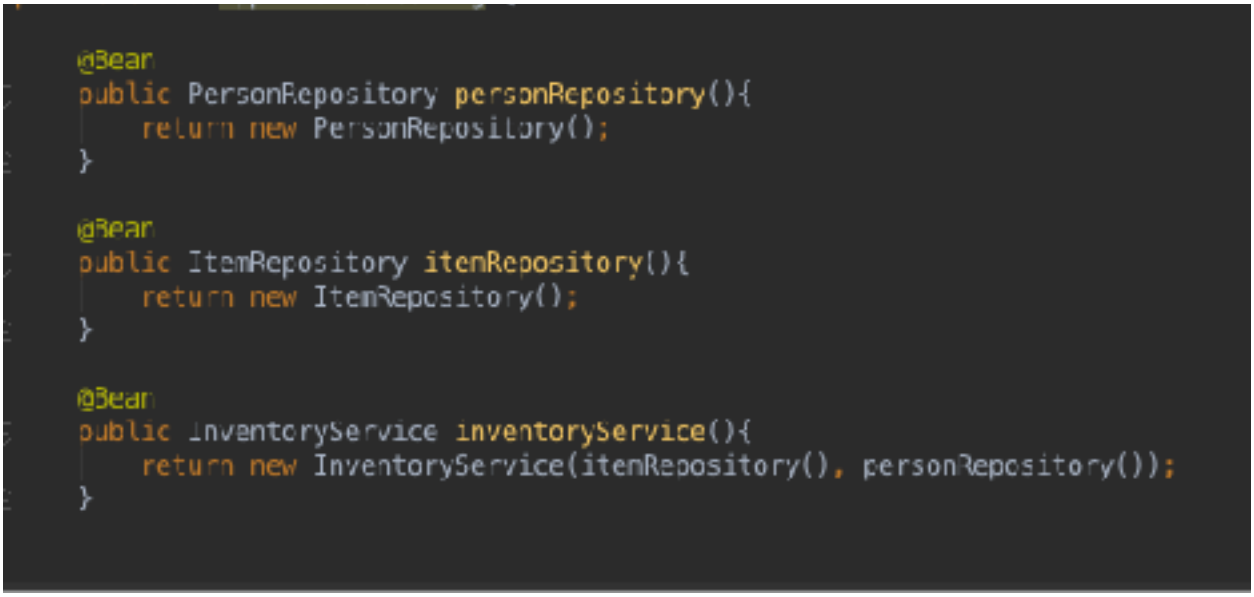
In the package com.frankmoley.talks.spring.intro.config create a class called ApplicationConfig.

Annotate the class with @Configuration to denote that this file is a Java Configuration class.

Create a bean annotated method for each of the repository classes in com.frankmoley.talks.spring.intro.data.repository

Use those beans in a bean annotated method for the InventoryService class in com.frankmoley.talks.spring.intro.service

When you are finished your beans should look similar to this:



Step 3: Prepare the Properties file

Create a file in /src/main/resources called *application.properties*

Annotate the ApplicationConfig class with @PropertySource("classpath:application.properties")

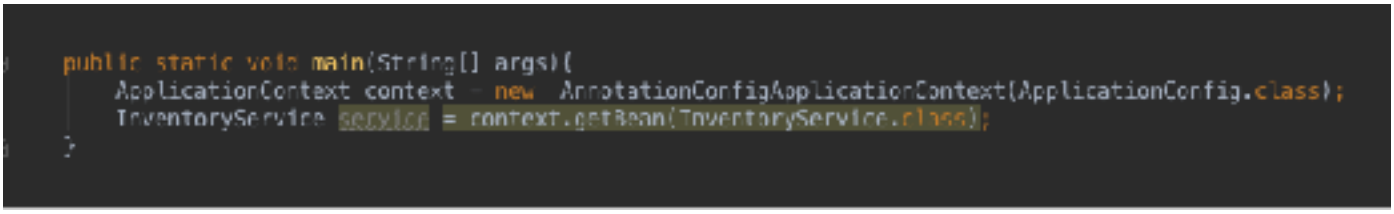
Step 4: Test the configuration

In the ApplicationConfig create a public static void main method

Create an instance of an ApplicationContext using the AnnotationConfigApplicationContext implementation of the interface passing it the ApplicationConfig.class as a parameter.

Now get an instance of the bean InventoryService from the context.

When completed it should look similar to this:



Now run the main method and it should exit normally with no errors. If you see a stack trace, check your configuration.

Take a break and we will start back up at the scheduled time!

Exercise 2

This exercise will leverage the information you have learned about Aspecting and help you create your first Spring Aspect. We will build a logging aspect to handle logging our service methods based on an annotation. We are starting with the end state of the previous exercise but we have added a log4j.properties file to the /src/main/resources folder for our logger.

Working with **/spring-intro-workshop/code/exercise_2/exercise project

Step 1: Add Dependencies

In this step we will add the dependancies for aspectJ and spring aop. Open the pom.xml and create dependency entries for the following. Note the compile scope doesn't need to be actually specified since it is the default.

Group	Artifact	Version	Scope
org.springframework	spring-sop	n/a	compile
org.aspectj	aspectjweaver	n/a	compile
org.slf4j	slfj4-api	n/a	compile
org.slf4j	slf4j-log4j12	n/a	runtime

Step 2: Create Aspect

In this step we will add our aspect and annotations to leverage.

Create a new package com.frankmoley.talks.spring.intro.aspect

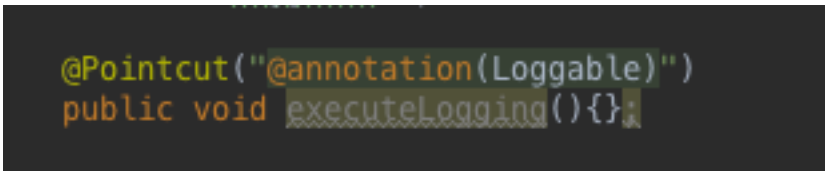
Create new annotation called Loggable in the above package. Add the following annotations to the annotation.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Loggable {
}
```

Create a new class in the aspect package called LoggableAspect. Annotate the class with @Aspect.

```
@Aspect
public class LoggableAspect {
```

Create a method of type public that returns void with no parameters and no body called executeLogging. Annotate the method with @Pointcut(“@annotation(Loggable)”).



Add a static final Logger to the class

Create a method called adviseLogging. The code below including the needed annotation is listed below:

```
@Around("executeLogging()")
  public Object adviseLogging(ProceedingJoinPoint joinPoint) throws Throwable{
     StopWatch stopWatch = new StopWatch();
     stopWatch.start();
     Object returnValue = null;
     try {
        returnValue = joinPoint.proceed();
        return returnValue;
     }finally {
        stopWatch.stop();
        StringBuilder message = new StringBuilder();
        message.append("method=").append(joinPoint.getSignature().getName()).append("|");
        message.append("time=").append(stopWatch.getTotalTimeMillis()).append("ms|");
        Object[] args = joinPoint.getArgs();
        if(null!=args && args.length>0){
           message.append("args={");
           boolean first=true;
           Arrays.asList(args).forEach(arg->{
              if(!first){
                 message.append("|");
              }
              message.append(arg);
           });
           message.append("}");
        }
        LOGGER.info(message.toString());
 }
```

Step 3: Utilize Aspect

Modify our UserService to return some stub data
In the UserService class **/exercise_2/exercises/src/main/java/com/frankmoley/talks/spring/intro/service add the following method

```
@Loggable
public User getUser(String emailAddress){
    User user = new User();
    user.setUsername("BigLebowski");
    user.setPersonId(UUID.randomUUID().toString());
    return user;
}
```

!Note if you don't want to pay homage to the The Dude, you can use any other stub data you want

Step 4: Configure the Application Context

In this step will configure the application context to engage the aspect and trigger it from a method in our service class.

In ApplicationConfig class add the annotation to enable AspectJ @EnableAspectJAutoProxy

```
@Configuration
@PropertySource("classpath:application.properties")
@EnableAspectJAutoProxy
public class ApplicationConfig {
```

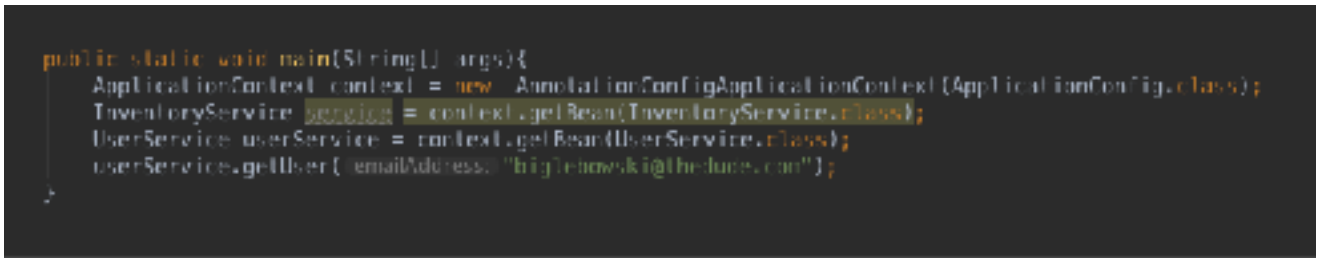
Create a bean for the LoggableAspect

```
@Bean
public LoggableAspect loggableAspect(){return new LoggableAspect();}
```

Create a bean for the UserService class

```
@Bean
public UserService userService(){return new UserService(personRepository());}
```

Now lets modify the main method to call the getUser method of the new UserService bean



Now execute the main method and look for the new logging statement.

AuditLogger:46 - method=getUser|time=29ms|args={biglebowski@thedude.com}

Take a break and we will start back up at the scheduled time!

Exercise 3

In this exercise we will utilize Spring Data in our application to serve data out of our service APIs.

Working with **/spring-intro-workshop/code/exercise_3/exercise project

While we are working with the same project, we have converted this to Component Scanning to make the future work easier and keep our config clean. If you want to continue without Component Scanning, please note the differences are simple to keep in place. You can run your existing main method at this point to confirm everything is working still.

Step 1: Add Dependencies

In this step we will add the dependancies for aspectJ and spring aop. Open the pom.xml and create dependency entries for the following. Note the compile scope doesn't need to be actually specified since it is the default.

Group	Artifact	Version	Scope
org.springframework.data	spring-data-jpa	n/a	compile
com.h2database	h2	n/a	compile
org.hibernate	hibernate-core	5.0.12.Final	compile
org.hibernate	hibernate-entitymanager	5.0.12.Final	compile
org.hibernate.common	hibernate-commons-annotations	5.0.1.Final	compile
org.hibernate.javax.persistence	hibernate-jpa-2.1-api	1.0.0.Final	compile

Step 2: Configure the Embedded Database & Transaction manager

We first need to annotate the class to enable JPA Repositories and Transaction Management. Use the @EnableJpaRepositories annotation and specify the base package of "com.frankmoley.talks.spring.intro.data.repository". Also use the @EnableTransactionManagement

```
@EnableJpaRepositories(basePackages = "com.frankmoley.talks.spring.intro.data.repository")
@EnableTransactionManagement
```

Now we will create a bean called dataSource of type javax.sql.DataSource. We will create this using the EmbeddedDatabaseBuilder class. We have provided a schema.sql and data.sql starter file to initialize the database with and the schema. Feel free to leverage it or modify the data as you feel more comfortable with.

```
@Bean
public DataSource dataSource(){
    EmbeddedDatabaseBuilder databaseBuilder = new EmbeddedDatabaseBuilder();
    return databaseBuilder.setType(EmbeddedDatabaseType.H2)
            .addScript("schema.sql")
            .addScript("data.sql")
            .build();
}
```

We will now create the EntityManagerFactory for hibernate persistence. We will leverage the ability to scan hibernate entities based on annotations, but if you prefer a hibernate persistence file you can definitely use it as well. This needs to be a Bean, so annotate it with the @Bean annotation. By convention, this bean needs to be named entityManagerFactory. If you wish to use another name, there is documentation available on how to rename it.

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

    LocalContainerEntityManagerFactoryBean entityManagerFactory = new LocalContainerEntityManagerFactoryBean();
    entityManagerFactory.setPackagesToScan("com.frankmoley.talks.spring.intro.cafe.entity");
    entityManagerFactory.setDataSource(dataSource());
    HibernateJpaVendorAdapter vendor = new HibernateJpaVendorAdapter();
    vendor.setShowSql(false);
    vendor.setGenerateDdl(false);
    vendor.setDatabase(Database.H2);
    entityManagerFactory.setJpaVendorAdapter(vendor);
    return entityManagerFactory;
}
```

The final bean we need to configure is the Transaction Manager.

```
@Bean
public JpaTransactionManager transactionManager() {
    JpaTransactionManager jpaTransactionManager = new JpaTransactionManager();
    jpaTransactionManager.setEntityManagerFactory(entityManagerFactory().getNativeEntityManagerFactory());
    return jpaTransactionManager;
}
```

Step 3: Annotate the Entity classes

Since our example is leveraging the javax.persistence annotations on our entity classes we must put them on the entity classes. In the package "com.frankmoley.talks.spring.intro.data.entity" open the ItemEntity class. Add the appropriate JPA annotations to the class. Note the column names from the schema.sql file in /src/main/resources.

```
@Entity
@Table(name="ITEM")
public class ItemEntity {
    @Id
    @GeneratedValue
    @Column(name="ITEM_ID")
    private long id;
    @Column(name="NAME")
    private String name;
    @Column(name="DESCRIPTION")
    private String description;
    @Column(name="MANUFACTURER")
    private String manufacturer;
    @Column(name="MODEL")
    private String model;
    @Column(name="QUANTITY")
    private int quantity;
    @Column(name="PERSON_ID")
    private long  personId;
```

Add a toString() method to this class so we can see the output later.

Do the same for the PersonEntity class.

```
@Entity
@Table(name="PERSON")
public class PersonEntity {
    @Id
    @GeneratedValue
    @Column(name="PERSON_ID")
    private long id;
    @Column(name="FIRST_NAME")
    private String firstName;
    @Column(name="LAST_NAME")
    private String lastName;
    @Column(name="EMAIL_ADDRESS")
    private String emailAddress;
```

Step 4: Modify the repositories for spring-data-jpa

With spring-data-jpa, we can leverage the power of the framework to create default methods for data access. Open the ItemRepository class in "com.frankmoley.talks.spring.intro.data.repository".

Convert the class to an interface and extend the CrudRepository interface. The template arguments specify the entity class and the type of the Id.

```
@Repository
public interface ItemRepository extends CrudRepository<ItemEntity, Long>{
```

Now we will create a simple method in this class to get the ItemEntities for a given personId.

```
List<ItemEntity> findByPersonId(long personId);
```

Now repeat the interface changes with the PersonRepository class

```
@Repository
public interface PersonRepository extends CrudRepository<PersonEntity, Long>{
}
```

Step 5: Test the changes

Now we can test the changes. Return to the ApplicationConfig class. Add the following code to the end of the main method.

```
PersonRepository personRepository = context.getBean(PersonRepository.class);
Iterable<PersonEntity> people =  personRepository.findAll();
people.forEach(System.out::println);
ItemRepository itemRepository = context.getBean(ItemRepository.class);
List<ItemEntity> items = itemRepository.findByPersonId(1L);
items.forEach(System.out::println);
```

Now run the main method and you will see data being output from the database and printed to the console.

Take a break and we will start back up at the scheduled time!

Exercise 4

In this exercise we will leverage Spring MVC to serve data out of a web page and a ReSTful endpoint. This example will need to be run in tomcat. We will be using Thymeleaf as our template language for the web page

Working with **/spring-intro-workshop/code/exercise_4/exercise project

Step 1: Dependencies and pom changes

We first need to package this application as a war instead of the default jar. To do this, under the project element, create an element called packaging and set it war.

```
<packaging>war</packaging>
```

Now we need to add the dependencies needed for this exercise.

Group	Artifact	Version	Scope
org.springframework	spring-web	n/a	compile
org.springframework	spring-webmvc	n/a	compile
org.thymeleaf	thymeleaf	n/a	compile
nz.net.ultraq.thymeleaf	thymeleaf-layout-dialect	n/a	compile
org.thymeleaf	thymeleaf-spring4	n/a	compile
com.fasterxml.jackson.core	jackson-databind	n/a	compile
javax.servlet	javax.servlet-api	n/a	compile

Step 2: Build out the service layer

In this step we will provide domain translation from the underlying database entity layer.

in the package com.frankmoley.talks.spring.intro.service open the UserService class. Create a method getAllUsers. Annotate the method with @Loggable. Get the collection from the personRepository findAll method. Iterate through the list and translate the PersonEntity to the User object using a String conversion on Id and a translate the emailAddress to a username. Add the List to a collection of Users and then return it.

```
@Loggable
public List<User> getAllUsers(){
    Iterable<PersonEntity> entities = this.personRepository.findAll();
    List<User> users = new ArrayList<>();
    entities.forEach(entity->{
        User user = new User();
        user.setPersonId(String.valueOf(entity.getId()));
        user.setUsername(entity.getEmailAddress());
        users.add(user);
    });
    return users;
}
```

Step 3: Build the web controller

We will start by creating a new class called AppController in com.frankmoley.talks.spring.intro.web.app. Annotate the class with @Controller and @RequestMapping responding at the root context.

```
@Controller
@RequestMapping(value="/")
public class AppController {
```

Now create a constructor that takes the UserService object and annotate it with @Autowired. Set the userService argument to a class level private final attribute of the same type and name

```
private final UserService userService;

@Autowired
public AppController(UserService userService){
    this.userService = userService;
}
```

Now we will respond to the to a Get request at the root using a method that returns String and takes a parameter of Model. This method is annotated with @GetMapping. We will add to the model an attribute of “users” with the value from the service call to getAllUsers. We will then return a String of “index” to point to the template we will create next.

```
@GetMapping
public String getHome(Model model){
    model.addAttribute( attributeName: "users", this.userService.getAllUsers());
    return "index";
}
```

Step 4: Build a simple template

In /src/main/resources create a directory called templates. In that directory create an HTML file called index.html. The HTML we will use is very simplistic.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>Title</title>
</head>
<body>
    <h1>Hello World</h1>
    <ul th:each="user: ${users}">
        <li th:text="${user.username}"></li>
    </ul>

</body>
</html>
```

Step 5: Configure the Web Container

Take note in this step of all the configuration we have created and will create as part of this step. As we progress into the next section, it will lead to some appreciation of how much Spring Boot saves you in writing code.

The first thing we will do is create a WebApplicationInitializer. Since servlets 3.0 we can replace the web.xml with a class file that extends this interface. So in the com.frankmoley.talks.spring.intro.config package, create a class called WebInitializer that extends WebAppliationInitializer. Implement the onStartup method of that interface using the following code.

```
public class WebInitializer implements WebApplicationInitializer{
    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        AnnotationConfigWebApplicationContext ctx =
                new AnnotationConfigWebApplicationContext();
        ctx.register(ApplicationConfig.class);
        ServletRegistration.Dynamic servlet =
                servletContext.addServlet( servletName: "dispatcher", new DispatcherServlet(ctx));
        servlet.addMapping( urlPatterns: "/");
        servlet.setLoadOnStartup(1);
    }
}
```

Now we will configure our ApplicationContext to resolve the Thymeleaf views.

In ApplicationConfig, create a bean method of type ClassLoaderTemplateResolver and name templateResolver.

```
@Bean
public ClassLoaderTemplateResolver templateResolver(){
    ClassLoaderTemplateResolver templateResolver = new ClassLoaderTemplateResolver();
    templateResolver.setPrefix("templates/");
    templateResolver.setCacheable(false);
    templateResolver.setSuffix(".html");
    templateResolver.setTemplateMode("HTML5");
    templateResolver.setCharacterEncoding("UTF-8");
    return templateResolver;
}
```

Now we need a template engine. Create a bean method of type SpringTemplateEngine called templateEngine. Reference the templateResolver bean in the templateEngine class.

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

Finally we need a View Resolver. Create a bean method of type ViewResolver named viewResolver and reference the templateEngine bean in the viewResolver class.

```
@Bean
public ViewResolver viewResolver() {
    ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
    viewResolver.setTemplateEngine(templateEngine());
    viewResolver.setCharacterEncoding("UTF-8");
    return viewResolver;
}
```

Go ahead and delete the main method, we no longer need it.

We can now deploy our application to tomcat. Using your IDE, start tomcat and deploy the war to it. Assuming your tomcat loads at port 8080, navigate in a browser to localhost:8080 and you should see the hello world view with all of the usernames listed from the embedded database.

Step 6: Create the ReSTful endpoint

Spring MVC uses similar technologies for serving a ReSTful endpoint. Create a class called ServiceController in com.frankmoley.talks.spring.intro.web.service. Annotate the class with @RestController and @RequestMapping("/api")

```
@RestController
@RequestMapping("/api")
public class ServiceController {
```

Just like with the AppController, inject the userService through the constructor via @Autowiring

```
private final UserService userService;

@Autowired
public ServiceController(UserService userService){
    super();
    this.userService=userService;
}
```

Now create a method that returns a List<User> objects called getUsers. It is a parameterless method annotated with @GetMapping("/users"). This method should simply return the data from the service call to getAllUsers

Once again we will start tomcat up and deploy the war file to it. Navigate this time to localhost: 8080/api/users and you should see a JSON payload of users from the database.

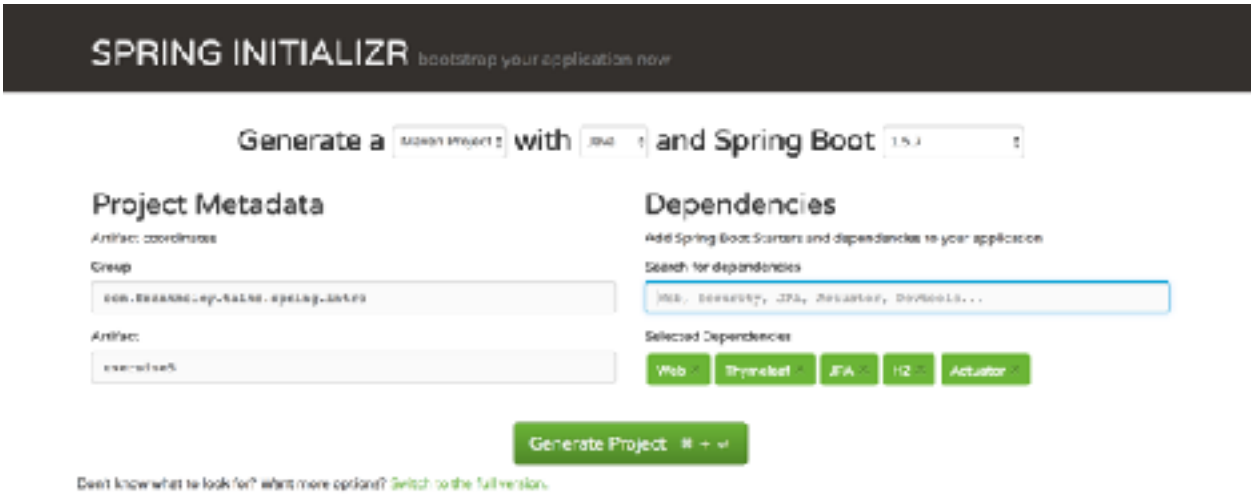
Take a break and we will start back up at the scheduled time!

Exercise 5
Please note, there is no exercise for this section as we will be leveraging start.spring.io to create the exercise for us. We also will simply copy files over to our application from the previous exercise, so no more coding for today!!!

Step 1: Build the application
Navigate in your browser to start.spring.io

Set a group of “com.frankmoley.talks.spring.intro”
Set an artifact name of exercise5

You can now either switch to full version or simply type them in the search box, but we are adding the following dependencies: web, thymeleaf, jpa, h2, actuator.



Now click generate project and allow it to download.

Step 2: Inspect
Take a look at the pom file. Notice how much simpler it is that in exercise4.
Take a look at the Exercise5Application class. This is our configuration class for this application, notice how there is no configuration documented. We won’t be adding any either, we will allow autoconfig to handle it all for us.

Step 3: Copy files over
Now you need to do a mass copy of all of you non-config class files from the previous exercise. You can maintain packages structure under the new package or you can simply dump everything into the root package, which is what I will do in the solution.

For the resources, you need to copy the 2 sql files to /src/main/resources and the index.html template to the /src/main/templates folder. We won’t need the log4j.xml anymore.

Step 4: Set the properties

With SpringBoot we do need to set a couple properties that we configured for hibernate before. in the /src/main/resources folder, in applicaiton.properties file add the following properties:

```
spring.jpa.hibernate.ddl-auto=none
logging.level.root=INFO
```

Step 5: Run the application

There are several ways to run this application. You will notice in the Exercise5Application class there is a main method, you can run it. You can also use the built in functions of STS or IntelliJ if you are using those IDEs to run the application. Finally you can leverage the executable jar we just build and using a mvn package command to build it then a java -jar exercise5.jar on it to run. Either way you choose, run the application and then navigate to localhost:8080 and localhost:8080/api/users and you should see the same pages.

That was a lot easier (granted all the code was written)

Take a break and we will start back up at the scheduled time!