Spring Boot Fundamentals

Course Overview

The Spring Boot Fundamentals course provides a comprehensive, developer's eye understanding of the Spring and Spring Boot ecosystem. It covers everything from basic Spring concepts, to advanced Spring Boot configuration, to creating and consuming REST applications connected to databases. It addresses testing at various degrees of resolution, and looks at framework support for monitoring Spring Boot applications. Along the way it explores various tricks and techniques to use Spring and Spring Boot effectively.

Introduction to Spring

- Capabilities of the modern Spring ecosystem.
- Spring v/s Spring Boot.
- Factory Pattern
- Dependency Injection and Inversion of Control
- Spring Application Context and Beans
- Bean Configuration
- DI with Spring
- Profiles
- Testing Spring Applications.

Introduction to Spring

- Structure of Enterprise Applications
  - Many jiggling parts (JPs)
  - The need for flexible relationships between the JPs
- Design for change
  - Hide JPs behind abstractions
  - Use dependency Injection
- Factory Pattern
- Spring is, at it’s core, a flexible and versatile Factory.
  - Manages the life cycles of JPs
    - When they get created
    - How many get created
  - Manages dependencies between JPs

Introduction to Spring

- Spring manages your objects for you.
- You tell it which objects you want it to manage by creating **bean** definitions
  - in XML files
  - Using annotations - **@Component, @Controller, @Configuration, @Bean** etc. etc.
- We will work with the annotation based configuration

Spring v/s Spring Boot

- So what's the difference between Spring and Spring Boot?
  - None
  - Spring Boot gives us a much smoother way to use Spring.
  - It's like being able to start your car with an electric starter rather than a hand crank.
  - But Spring is still the engine that's actually getting you places.
- Much more on Spring Boot later.

Key Spring Concepts

- **Bean**s
  - All objects you want Spring to manage have to be declared as **bean**s.
    - 99.99% true
- **ApplicationContext**
  - The factory. A Spring application always starts up by creating an Application Context.
  - Referred to as **the context** or **the container.**
  - The context class reads your configuration files and creates and wires up your objects.
  - You need to tell it where to find your configuration.
  - You may not always need to interact with it directly, but it is an important object to be aware of.

Key Spring Concepts

- Dependency Injection
  - An object does not decide who it’s collaborators are.
  - “Don’t call us, we’ll call you”
  - The decision about collaborators is made externally, often by a framework, based on configuration you provide.
  - In our case, the framework is Spring.
  - Spring provides several different ways to tell it what to inject where.
    - Using annotations, Spring does the wiring reflectively
      - @Autowired, @Resource, @Inject
    - Using @Bean methods, you do the wiring and return to Spring a fully wired bean.

Configuration Annotations

- Spring configuration annotations
  - **@Component –** Used on a Class to tell Spring to create a bean for that class.
    - **@Service, @Repository, @RestController** etc. **–** for specific types of components.
  - @**Configuration -** Used on a Class to tell Spring that it contains bean definition methods.
  - **@Bean –** Used on a method, usually in an **@Configuration** class. Spring will call the method when the context is being created. You write code in the method to create your object and initialize in any way you want, and then return it to Spring. Your object then becomes a **bean.**
    - sometimes called a “Producer Method” in other frameworks.

@Configuration and @Bean

- The **@Configuration** annotation signal to Spring that the class contains bean definition methods.
- You create methods in the class annotated with **@Bean.**
- Spring calls these methods when it is creating the context.
- You create your application objects and return them to Spring all nicely wired up.
- You use a special syntax to refer to beans created with other **@Bean** methods if you need to inject them into the object you are creating.

@Configuration and @Bean

```
@Configuration
@ComponentScan({"ttl.larku.service", "ttl.larku.dao"})
public class LarkUConfig {

    @Autowired
    private Environment env;

    @Bean
    @Profile("development")
    public BaseDAO<Student> studentDAO() {
        return new InMemoryStudentDAO();
    }

    @Bean(name = "studentDAO")
    @Profile("production")
    public BaseDAO<Student> studentDAOJpa() {
        return new JpaStudentDAO();
    }

    @Bean
    public StudentService studentService() {
        StudentService ss = new StudentService();
        ss.setStudentDAO(studentDAO());

        return ss;
    }
}
```

Scan these packages for @Component and @Configuration classes

Can specify that a bean will be created only if a specific profile is active.

Dependency injection

@Component et. al.

```
//@Component
@Service
public class StudentService {

    @Autowired
    private BaseDAO<Student> studentDAO;

    …
}
```

@Component is the base annotation, but prefer the more specific annotations where appropriate.

@Service for Service classes
@Repository for DAO like classes

Dependency Injection by type (mostly)

Dependency Injection Annotations

- **@Autowired**
  - tries to match by type
  - then tries to match by **@Qualifier** annotations
  - then tries to match by name
- **@Inject** has the same behaviour as @Autowired
- **@Resource**
  - tries to match by name
  - then tries to match by **@Qualifier** annotations
  - then tries to match by type
- **@Qualifier** - Used to help fine tune dependency in case of clashes

@Autowired

```
@Component
public class TricksWithAutoWired {
    //There better be 1 and only 1 bean of type CourseService.
    //Or, Just to make things confusing, if there is
    //more than 1 bean of type CourseService, then
    //only 1 of them should have the name courseService
    @Autowired
    private CourseService courseService;

    //Two features in one example.
    //Won't fail if nothing matches.
    //And will bring in *all* beans of
    //type CourseService in the context.
    @Autowired(required = false)
    private CourseService[] courseServices;
}
```

@Autowired

```
@Component
class Autowired2 {
    //Useful and safe for injecting
    //framework objects that we *know* there
    //will be only one of.
    @Autowired
    private ApplicationContext context;

    //Can qualify which bean to use
    //in case of clashes
    @Autowired
    @Qualifier("us-east")
    private BaseDAO<Student> studentDAO;
}

@Component
@Qualifier("us-east")
class DAOForEast extends InMemoryStudentDAO {

}
```

Constructor Injection

```
@Component
class ConstructorInjectionDemo {
    //@Autowired
    private final CourseService courseService;

    //Prefer constructor injection to field injection if possible.
    //Can use the @Autowired annotation to specify which
    //constructor Spring should use.  All arguments passed
    //in to the constructor should be beans.
    public ConstructorInjectionDemo(CourseService courseService) {
       this.courseService = courseService;
    }
}
```

Can make fields final with constructor injection

@Resource

```
@Component
class TricksWithResource {
    //There better be a bean named courseService.
    //Or, to make things confusing, if there is
    //more than one bean named courseService, than
    //only one of them should be of type CourseService
    @Resource
    private CourseService courseService;

    //This would work either if there was a bean
    //named "cs" or a bean of type CourseService
    //called something else.
    @Resource
    private CourseService cs;

    //Always use a name with @Resource
    @Resource(name = "studentService")
    private StudentService studentService;
}
```

@Resouce or @Autowired?

- As a general rule:
  - Use @Autowired to inject beans by **type**
    - Use **@Primary** or **@Qualifier** or **@Profile** to resolve clashes
  - Use @Resource to inject beans by **name**
    - And **always** specify the name
      - @Resource(name = “fancyService”)

Testing Spring applications

- Several ways to test Spring applications
- The big issue is what to do about the dependencies of the classes you want to test
- You can (and should) write true *Unit* tests which just test the code of the class under test. In this case you need to using a mocking library like Mockito to mock the dependencies. We will encounter Mockito later in the course
- If you need to test a class with it's dependencies, then you need to get Spring into the picture to make the object and wire it up properly.
- For JUnit5 use
  - **@Extension(SpringExtension.class)**
- For JUnit4 use
  - **@RunWith(SpringRunner.class)**

Testing Spring applications

- By default, Spring will create the context once for for a test run.
  - Various ways to configure exactly what goes into the context.
  - We will see several examples in the code
- Spring can cache contexts between tests. It uses various keys for caching, e.g.
  - Configuration classes used to create the context.
  - Active profiles set for the test.
  - etc.
- If it finds that two tests use the same configuration, it reuses a previously cached context if available.
- So it may be worthwhile to run a set of tests with a common context configuration rather than paring down the classes for each test.
  - Examples in code.

Testing Spring applications

Allow Spring to set up the class before the test runs.

Tell Spring where to find your bean definitions.

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = LarkUConfig.class)
public class StudentServiceTest {

    @Autowired
    private StudentService studentService;

    @BeforeEach
    public void setup() {
        studentService.clear();
    }

    @Test
    public void testCreateStudent() {
        Student newStudent =
         studentService.createStudent(name1, phoneNumber1, Student.Status.FULL_TIME);

        Student result = studentService.getStudent(newStudent.getId());

        assertTrue(result.getName().contains(name1));
        assertEquals(1, studentService.getAllStudents().size());
    }
}
```

So where does Spring Boot come in?

- Advantages of using Spring Boot
- start.spring.io
- Tuning Spring Boot with application.properties
- Externalizing configuration - @ConfigurationProperties
- Using Spring Boot Testing goodness.

So where does Spring Boot come in?

- Remember the JPs?
- So far we have built an application with around 6 or 7 of them.
- We have provided configuration for all of them.
- We have also provided library dependency information for all the various libraries we need in our build system configuration files (pom.xml, build.gradle etc)
- Now imagine many 10’s of beans of various types
  - Our own application classes, classes from other libraries, web servers, connections to databases, connections to messaging systems, classes to monitor the state of our application. Etc. etc.
- Wouldn’t it be nice if we had a way to auto configure some of this?
- Enter Spring Boot, stage left.

Spring Boot

- Think of Spring Boot as an uber configurer.
  - It helps you specify library dependencies using **starter** packages which can reduce significantly the number of things you need to put into your build configuration files.
  - At run time, it looks at the library dependencies you have specified. Based on them it can **auto configure** infrastructure objects for you:
    - Start web servers if you have specified web libraries.
    - Create connections pools to databases and messaging systems if you have specified those libraries.
    - Expose monitoring information if you have asked it to.
    - Etc. etc.
  - These two capabilities together make Spring Boot a much more pleasant way of configuring and starting a complicated Spring application than having to do all the configuring and starting yourself.

Creating a Spring Boot application

- Easiest way is to use the starter mechanism from spring.io
- Over the web
  - **http://start.spring.io**
- Or, if you have the Ultimate Edition of IntelliJ, from within the IDE
    - File → New Project → Spring Initializer

Example Pom.xml

```
<?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>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.1.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>ttl.larku</groupId>
    <artifactId>SpringBoot</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>SpringBoot</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
     …
    </dependencies
</project
```

Parent pom sets up common configuration and dependency versions based on the version of Spring Boot we are using.

Example Pom.xml

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

Web starter will bring in Spring MVC libraries and a server, tomcat by default. Server will be started when the application is

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

Test libraries (JUnit 5, Mockito, JsonPath, etc.) included automatically.

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

Expose metrics for your app

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

JDBC, JPA, Hibernate, etc. DataSources will be created based on your specifications.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

H2 embedded database. Spring Boot can start the database and create and initialize the tables at start up.

Example application.properties

```
#Server port - default is 8080
server.port=8080

#Logging
logging.level.org.springframework=debug
logging.level.org.hibernate=INFO

#Profile
spring.profiles.active=development

##########################Actuator#############################
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

@SpringBootApplication

- Annotation for your main application class.
- 3 annotations bunched in one
  - @**Configuration**
  - **@ComponentScan**
  - **@EnableAutoConfiguration**
- By default, your **@SpringBootApplication** class should be at the root of the package hierarchy that you want to Spring to scan to look for your @Configuration and @Component files.

@SpringBootApplication

```
package ttl.larku;

@SpringBootApplication
public class SpringBootApp {
    public static void defaultmain(String[] args) {
        SpringApplication.run(SpringBootApp.class, args);

    }
}
```

Spring will scan for @Configuration and @Component classes starting from this package.

Initializing Beans with Properties

- Scenario – you need to initialize simple properties of a bean with values that can change from run to run.
- Obvious answer is to put the values in a properties file and read them at initialization time.
- How can we make Spring do this for us when it creates our bean?
  - For just a few properties you can use **@Value** along with the **Sp**ring **E**xpression **L**anguage (SPEL).
  - For more properties, or if you have to inject the properties in several beans, an **@ConfigurationProperties** class may be the answer.
  - Also very useful if you want to set properties on a class that you don’t own.
  - Examples on next few pages, and in the code.

Properties with @Value

```
@Service
public class AtValueDemoService {

    //Read a property from the Environment
    //using "${...}"
    @Value("${spring.profiles.active}")
    private String profiles;

    //Run code and do comparisions using "#{...}
    @Value("#{'${spring.profiles.active}' == 'development'}")
    private boolean isDevelopment;

    //Call a method on another bean.
    @Value("#{connectionService.getHost()}")
    private String host;

    …
}
```

@ConfigurationProperties

```java
@Component
@ConfigurationProperties("ttl.connection-service")
@PropertySource("classpath:larkUContext.properties")
public class ConnectionServiceProperties {

    private String host;
    private int port;
    private Duration timeOut;
    private int retriesOnTimeout;

    //getters, setters etc.
}
```

Property prefix. Must use **canonical** property name, I.e lower case kebab cased.

Property Source

```
#larkUContext.properties
ttl.connectionService.host=xyz.com
ttl.connectionService.port=9999
ttl.connectionService.timeout=10s
ttl.connectionService.retriesOnTimeout=3
```

```java
@Service
public class ConnectionService {

    @Autowired
    private ConnectionServiceProperties props;

    public void makeConnection() {
        int retries = props.getRetriesOnTimeout();
        String host = props.getHost();

       //Use properties
    }
}
```

Can Autowire the Properties class where we want to use the properties.

Using the properties.

Configuring a class we don’t own

```
@Configuration
@PropertySource({"classpath:/larkUContext.properties"})
public class LarkUConfig {

  /**
   * You can use @ConfigurationProperties on an
   * @Bean method. Can be useful when you want
   * to initialize classes you don't
   * own.
   *
   * @return
   */
  @Bean
  @ConfigurationProperties("ttl.stwdo.config")
  public ServiceThatWeDontOwn serviceThatWeDontOwn() {
    return new ServiceThatWeDontOwn();
  }
}
```

#larkUContext.properties

```
#Configuration Properties for
#ServiceWeDontOwn
ttl.stwdo.config.timeout=10s
ttl.stwdo.config.countDown=false
```

```
//Our code
@Autowired
private ServiceThatWeDontOwn stwdo;

stwdo.doServicyTypeThing();
```

```
//We don’t own this class
public class ServiceThatWeDontOwn {

    private Duration timeout;
    private boolean countDown;

    //Getters and Setters and
    public void doServicyTypeThing(){
         ...
    }
}
```

Testing with SpringBoot

- **@SpringBootTest**
  - The issues with contexts and caching are the same as we have seen earlier. Because Spring Boot tests are Spring tests.
  - All the techniques we have seen earlier still apply.
  - @SpringBootTest gives you some several extra bells and whistles.
    - e.g., you don’t need the @ExtendWith annotation.
- Spring Boot also has some convenient shortcuts, called **slice** tests, for testing some specific *slices* of the application, e.g.
  - @DataJpaTest – testing Spring JPA repositories
  - @WebMvcTest – testing *only* Web controllers
- For testing controllers there is **MockMVC**
- Examples in the SpringBoot project

Testing with SpringBoot

```
//@ExtendWith(SpringExtension.class)
```

Not needed anymore, as @SpringBootTest includes it.

```
//Different ways of creating the Context.  From Most Expensive to
// Not So Expensive.
/**********************
 //This is the most expensive way to make the context.  The entire
 //Spring Context will be created. About 104 beans, at this writing.
 **********************/
//@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.NONE)

/**********************
 //This next approach will just create the beans in the given
 //configuration classes. //About 13 beans at this writing.
 //Note that we need to include StudentService.class here only because
 //we have declared it with @Service, and no classpath scanning
 //will take place if we use this approach.  Alternative would
 //be to declare the StudentService in LarkUConfig.
 **********************/
//@SpringBootTest(classes = {LarkUConfig.class, LarkUTestDataConfig.class, StudentService.class})

/*****************************
 * Finally, we can simply pull in the exact classes we need
 * for the test.  This will still pull in some other beans.
 * About 10 beans at this writing.
 **********/
@SpringBootTest(classes = {StudentService.class, InMemoryStudentDAO.class,
                                                 ClassThatWeDontOwn.class})
public class StudentServiceTest {
    …
}
```

Issues with Test Context Caching

- There is a flip side to the technique of only include the classes you need for each test.
- Spring caches application contexts that it creates for your tests.
- If possible, it tries to reuse a context over creating a new context.
- Contexts are cached using several aspects of the configuration as keys including
  - configuration classes
  - profiles
  - etc.
- So, instead of specifying exact classes, it may be better to add all the classes necessary for a suite of tests.

Issues with Test Context Caching

- Specifying exact classes for each tests results in Spring creating a new context for each test. Maybe with fewer beans, but an expensive operation none the less.
- It may be better to amortize that cost among a group of related tests.
  - Specify the classes needed for all the tests when running them.
  - This will allow Spring to create the context the first time, and then reuse it because the configuration specified is the same.
- You can do this by either organizing your actual @Configuration files so they are testable as a bundle.
- Or by creating an **@TestConfiguration** just for the tests.
- Examples in the **ttl.larku.service.reg** test package

Spring REST Services

- What is REST?
- Controllers and RestControllers
- Configuration
- Content Negotiation
- Controller Mappings
- Customizing Controller Responses
- Consuming REST Services
- Testing support from Spring

Spring REST Services

- REST
  - **Re**presentational **S**tate **T**ransfer
- The state being transferred is often referred to as a **resource.**
- Uses HTTP extensively
  - HTTP methods specify what the client wants the server to do
    - GET – retrieve a resource
    - DELETE – delete a resource
    - PUT – update a resource
    - POST – usually insert a new resource, but POST can be used to do whatever.
  - HTTP Headers to carry other information
    - Accept, Content-Type
    - Authentication
    - Etc. etc.

Spring REST Services

- Spring MVC REST Annotations
  - @RestController – declare this class to be a RestController
  - RequestMappings
    - @GetMapping
    - @PostMapping
    - @PutMapping
    - @DeleteMapping
    - @RequestMapping – Let’s you do all/any of the above
  - @PathVariable – capture parts of a URL

Rest Controller

Tells Spring that this class has methods that should be mapped to Http requests.

Root Url.

Path Variable

```
@RestController
@RequestMapping("/adminrest/student")
public class StudentRestController {

    @Autowired
    private StudentService studentService;

    @GetMapping("/{id}")
    public ResponseEntity<?> getStudent(@PathVariable("id") int id) {
        Student s = studentService.getStudent(id);
        if (s == null) {
            return ResponseEntity.badRequest().body(new RestResult(RestResult.Status.Error,
                    "Student with id: " + id + " not found"));
        }
        return ResponseEntity.ok(new RestResult(s));
    }
}
```

Return not only the body, but also an appropriate Http Status Code.

Testing Controllers

- We can create Unit tests.
  - Create the class as a Java class.
  - Mock out any dependencies.
- This does not test the Spring MVC machinery.
  - Request mappings.
  - Response codes.
  - Data conversions.
  - etc.

Slice Test - @WebMvcTest

- A first step into Integration testing is a so called *slice* test, which sets up only a part of the context to exercise a particular feature.
- **@WebMvcTest –** To test *only* your controllers.
  - No application beans will be created.
  - Which means they will need to be mocked out.
  - Spring's **@MockBean** is an alternative and simpler way to do this than Mockito
  - Example in **StudentRestControllerSliceTest**

Testing Controllers - MockMVC

- The next step into Integration testing is to use the MockMVC library.
- Creates mock http requests that look to the MVC library like the real thing.
  - The requests go through all the same processing as a real request.
    - (99.9%)
  - But without having to run a real server.
  - Creates your entire application context.
- The library allows you to set up expectations about the results
  - status codes
  - returned content type
  - assertions about the returned content
  - etc.
- Example in **StudentRestControllerMvcTest.**

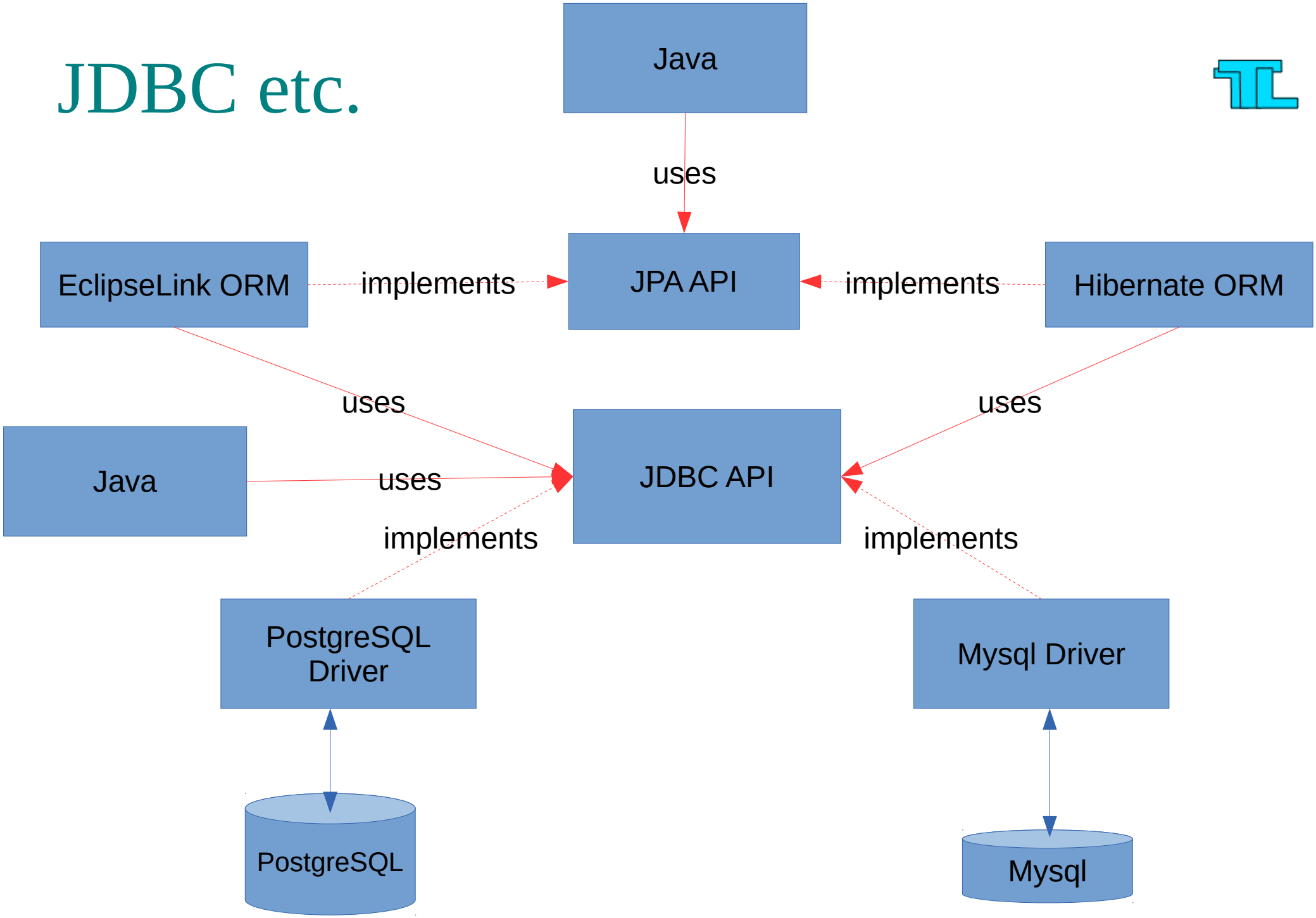
Testing with a real server

- Moving on up the chain of Integration testing, you can have SpringBootTest run a real server for the test
  - The server can run on a random port.
  - **@LocalServerPort** annotation to inject random port into your test.
- Use a Rest client for testing.
  - Spring has RestTemplate and TestRestTemplate.
- Example in **RestClientSpringTest**

Spring Data JPA

- Overview of JDBC and JPA/Hibernate
- Spring Data features
- DataSource configuration
- Initializing Databases
- Spring Data Repositories
- Testing support from Spring

JDBC etc.



Spring Boot and databases

- As expected, Spring Boot can do some lifting for us when interacting with databases.
- AutoConfiguration can set up DataSources and connection pools for you based on available properties.
- You can ask Spring Boot to start and initialize databases on start up.
  - Useful for embedded databases.
  - Useful for testing.

DataSource configuration

```
#H2
spring.datasource.url=jdbc:h2:mem:LarkUDB
spring.datasource.username=larku
spring.datasource.password=password
spring.datasource.driverClassName=org.h2.Driver
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true
```

Required properties for creating a JDBC DataSource.

Java Persistence API

- An abstraction on top of **O**bject **R**elational **M**apping (ORM) tools like Hibernate and EclipseLink.
- The actual ORM tool you use is called the *provider* rather than a driver.
  - You do need to have the appropriate JDBC drivers also available for the provider to use.
- JPA allows you to use annotations to map Java objects to tables and columns in a relational database.
- The idea being that you work and think in objects and let the JPA provider deal with the grunt work of moving data in and out of the database.

JPA Entity

```
@Entity
public class Student {

    public enum Status {
        FULL_TIME,
        PART_TIME,
        HIBERNATING
    };

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private String phoneNumber;

    @Enumerated(EnumType.STRING)
    private Status status = Status.FULL_TIME;

    //constructors, getters and setters, etc.
}
```

Required. Tells Provider to map this class. Table name by default is Class name.

Required. Identifies the primary key.

Whether/How to generate the key. Here we use the Identity mechanism of the database.

Depending on the type, you may need other annotations. Here we specify that enums should be stored as Strings in the DB

All fields will be mapped to columns. Mapping strategy can be configured.
Here, by default, the fields will be mapped to columns with the same name.

Working with JPA

- JPA Providers use JDBC to interact with the DB
  - JDBC driver needs to be on the classpath
  - DataSource needs to be configured.
- Two JPA concepts:
  - PersistenceUnit
    - Encapsulates your configuration information.
    - API class is **EntityManagerFactory**.
    - Configuration needs to be in a file called persistence.xml.
  - PersistenceContext
    - The thing you use to actually interact with the Database.
    - API class is **EntityManager.**
    - Created by an EntityManagerFactory.
- Transactions are vitally important when using JPA
  - All changes to the database *have* to be done in a transaction.

Spring JPA Repositories

- In Spring terminology a Repository is roughly equivalent to a DAO.
- Spring can implement a Repository (DAO) for a particular JPA entity for you automatically.
  - All you do is create an interface that extends a Spring Repository interface.
  - Spring will create the implementation at runtime.
  - The implementation gives you access to all the standard CRUD operations.
  - You can create additional operations by adding methods to your interface that follow particular naming conventions.
  - https://docs.spring.io/spring-data/jpa/docs/2.3.4.RELEASE/reference/html/#jpa.repositories

Spring and JPA

- Spring/ Spring Boot can
  - Create the PersistenceUnit for you based on properties.
    - No need for a persistence.xml file.
  - Inject the EntityManagerFactory and/or the EntityManager into your code.
  - Start transactions for you automatically based on configuration information
    - Usually via the @**Transactional** annotation.
- Example of using EntityManger in **SpringDB** project in **ttl.larku.dao.jpahibernate.JPAStudentDAO.**
- Example of Repository in **repository.[Simple]StudentRepo**

Testing database interaction

- Spring can
  - Start and stop databases
  - Create and populate databases at configurable times in the test lifecycle
    - once per test suite
    - once per test class
    - once per test method
    - etc.
  - Manage Transactions
    - By default, rolls back any interaction with the DB in a test method, leaving the DB in the initial state for the next method. Usually what you want.
  - Several examples in the tests for the **ttl.larku.dao** classes in the **SpringDB** project.

Testing JPA Repositories

- Spring has a *slice* test for testing JPA Repositories
  - **@DataJpaTest** instead of @SpringBootTest.
  - Will only load what is necessary to test the repository.
- To use @DataJpaTest with code that uses an EntityManager explicitly, you also need to include all your application classes in the context.
  - Can be done using the **@Import** annotation and pointing at all your configuration classes.
  - Or by creating a TestConfiguration class as we saw with the Spring Boot Tests earlier on.
- Example in the **SpringDB** project in **ttl.larku.dao.StudentDAOTest** and **StudentRepoTest**

The End