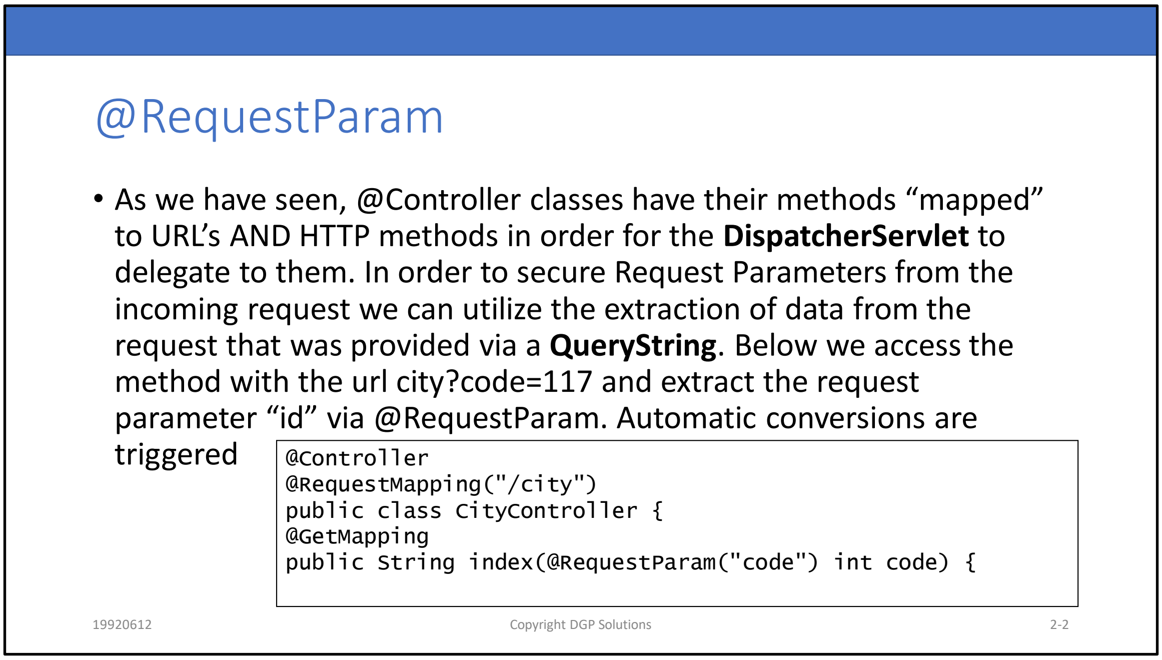
2. Controllers

DGP Solutions



@RequestParam

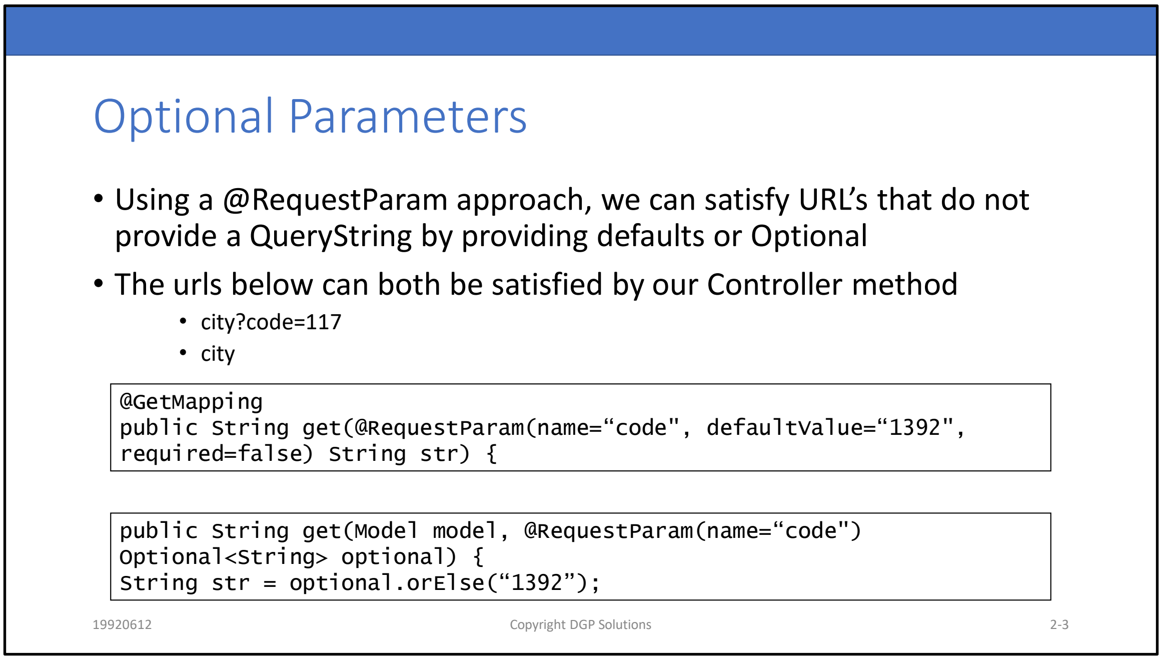
- As we have seen, @Controller classes have their methods “mapped” to URL’s AND HTTP methods in order for the **DispatcherServlet** to delegate to them. In order to secure Request Parameters from the incoming request we can utilize the extraction of data from the request that was provided via a **QueryString**. Below we access the method with the url city?code=117 and extract the request parameter “id” via @RequestParam. Automatic conversions are triggered

```
@Controller
@RequestMapping("/city")
public class CityController {
@GetMapping
public String index(@RequestParam("code") int code) {
```

Notice the concatenation of URL via the @RequestMapping annotation on the controller itself (such an annotation is not necessary as we can decide to annotated at the method level alone)

We can also use the @RequestMapping at the method level as well, however it is more verbose as we must specify the HTTP method

```
@RequestMapping(path="/single”, method=RequestMethod.GET)
public String getSingleEmplParam(@RequestParam(name="id")
long id) {
```

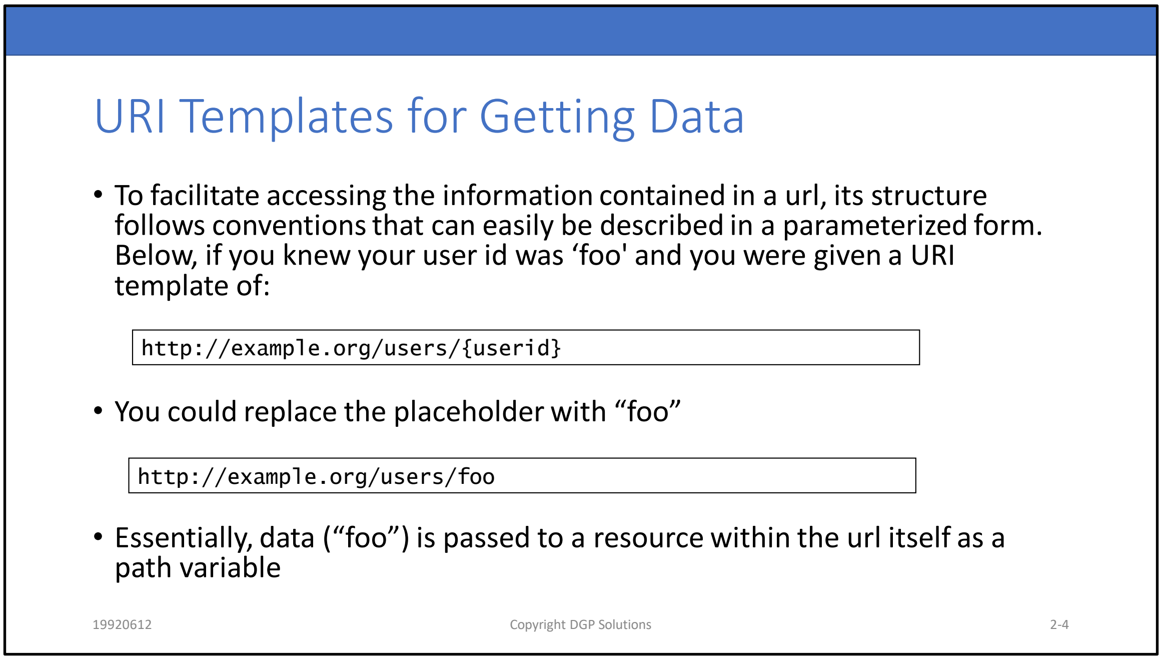


Optional Parameters

- Using a @RequestParam approach, we can satisfy URL’s that do not provide a QueryString by providing defaults or Optional
- The urls below can both be satisfied by our Controller method
    - city?code=117
    - city

```
@GetMapping
public String get(@RequestParam(name=“code", defaultValue=“1392",
required=false) String str) {
```

```
public String get(Model model, @RequestParam(name=“code")
Optional<String> optional) {
String str = optional.orElse(“1392”);
```



URI Templates for Getting Data

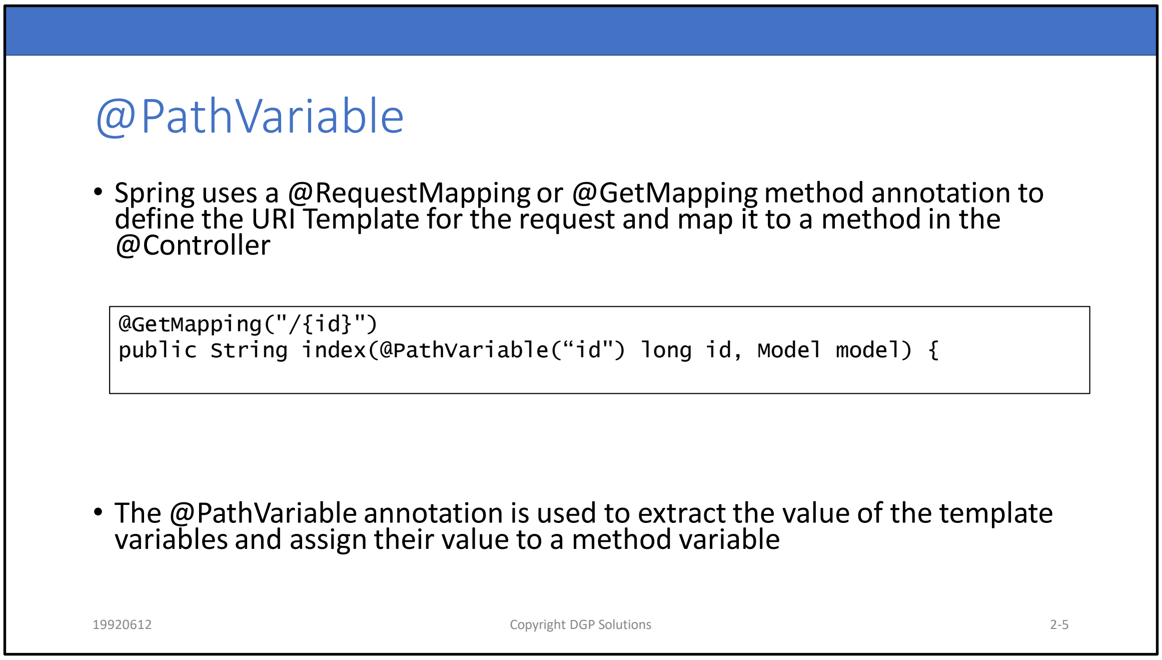
- To facilitate accessing the information contained in a url, its structure follows conventions that can easily be described in a parameterized form. Below, if you knew your user id was ‘foo' and you were given a URI template of:

```
http://example.org/users/{userid}
```

- You could replace the placeholder with “foo”

```
http://example.org/users/foo
```

- Essentially, data (“foo”) is passed to a resource within the url itself as a path variable

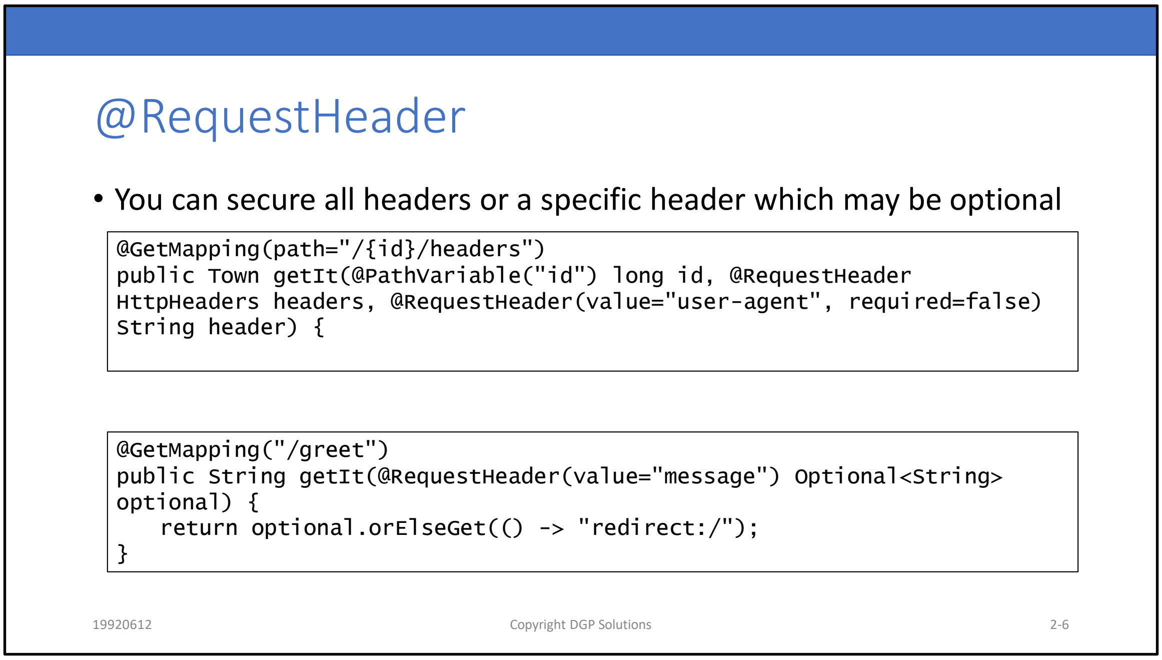


@PathVariable

- Spring uses a @RequestMapping or @GetMapping method annotation to define the URI Template for the request and map it to a method in the @Controller

```
@GetMapping("/{id}")
public String index(@PathVariable(“id") long id, Model model) {
```

- The @PathVariable annotation is used to extract the value of the template variables and assign their value to a method variable

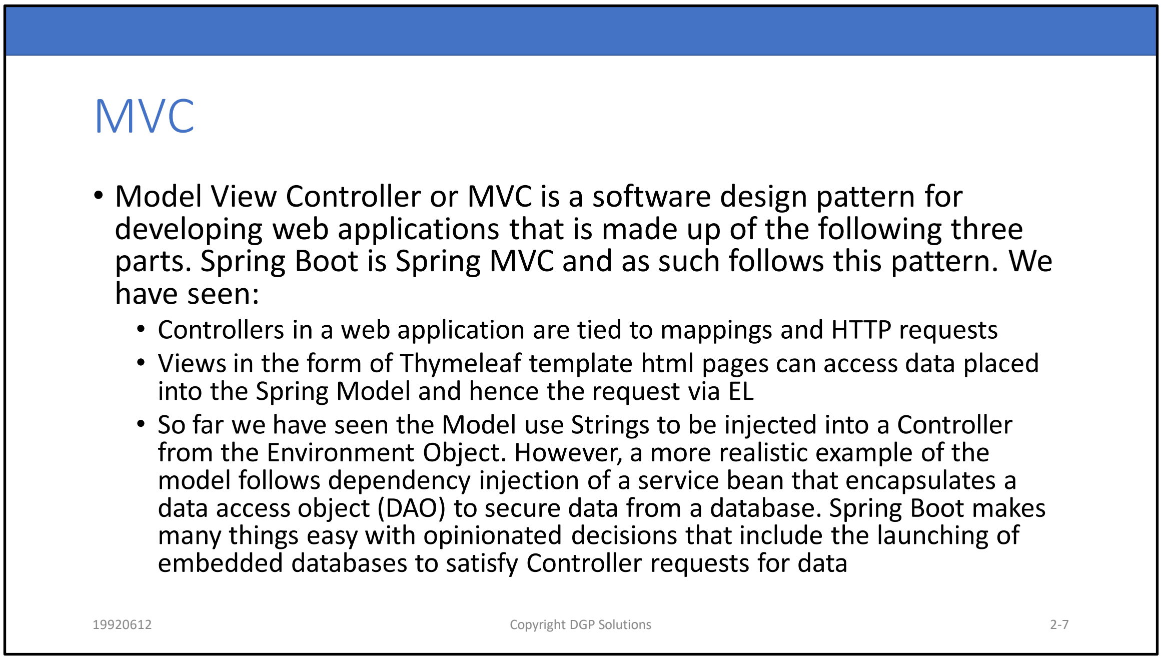


@RequestHeader

- You can secure all headers or a specific header which may be optional

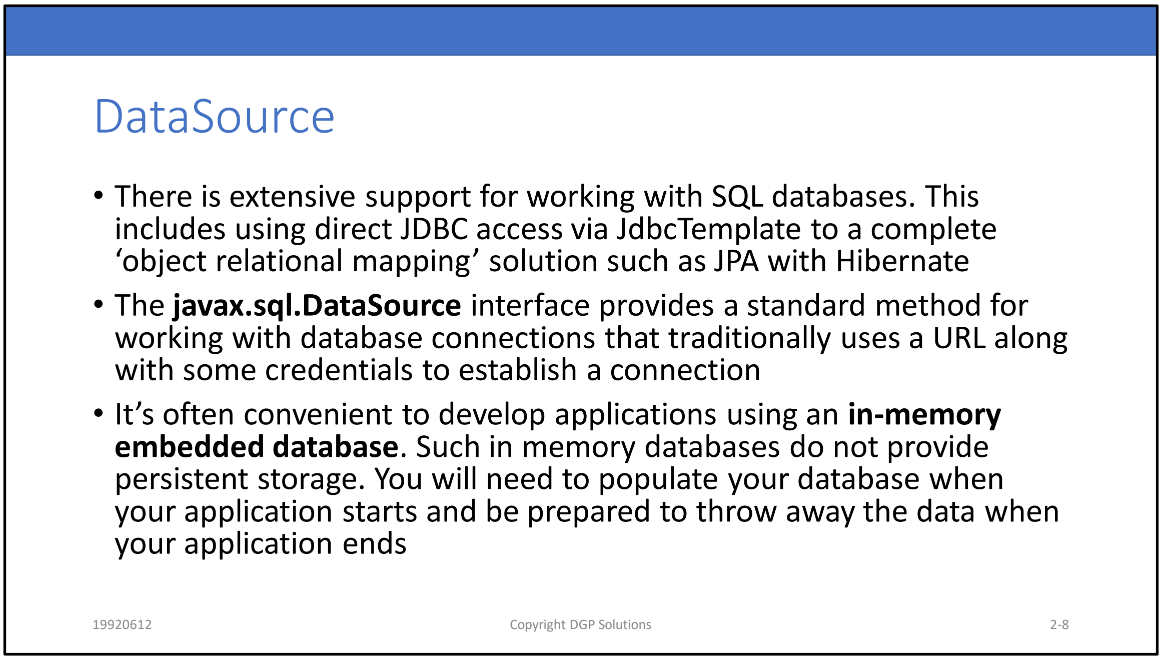
```
@GetMapping(path="/{id}/headers")
public Town getIt(@PathVariable("id") long id, @RequestHeader
HttpHeaders headers, @RequestHeader(value="user-agent", required=false)
String header) {
```

```
@GetMapping("/greet")
public String getIt(@RequestHeader(value="message") Optional<String>
optional) {
    return optional.orElseGet(() -> "redirect:/");
}
```



MVC

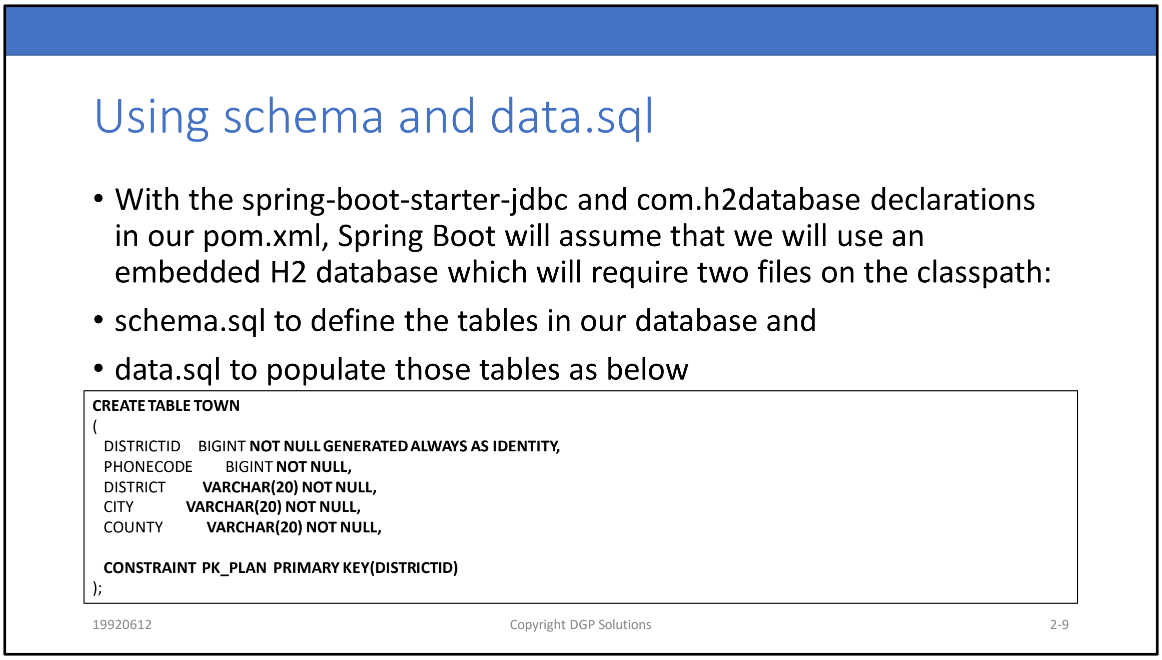
- Model View Controller or MVC is a software design pattern for developing web applications that is made up of the following three parts. Spring Boot is Spring MVC and as such follows this pattern. We have seen:
  - Controllers in a web application are tied to mappings and HTTP requests
  - Views in the form of Thymeleaf template html pages can access data placed into the Spring Model and hence the request via EL
  - So far we have seen the Model use Strings to be injected into a Controller from the Environment Object. However, a more realistic example of the model follows dependency injection of a service bean that encapsulates a data access object (DAO) to secure data from a database. Spring Boot makes many things easy with opinionated decisions that include the launching of embedded databases to satisfy Controller requests for data



DataSource

- There is extensive support for working with SQL databases. This includes using direct JDBC access via JdbcTemplate to a complete ‘object relational mapping’ solution such as JPA with Hibernate
- The **javax.sql.DataSource** interface provides a standard method for working with database connections that traditionally uses a URL along with some credentials to establish a connection
- It’s often convenient to develop applications using an **in-memory embedded database**. Such in memory databases do not provide persistent storage. You will need to populate your database when your application starts and be prepared to throw away the data when your application ends

19920612 Copyright DGP Solutions 2-8



Using schema and data.sql

- With the spring-boot-starter-jdbc and com.h2database declarations in our pom.xml, Spring Boot will assume that we will use an embedded H2 database which will require two files on the classpath:
- schema.sql to define the tables in our database and
- data.sql to populate those tables as below

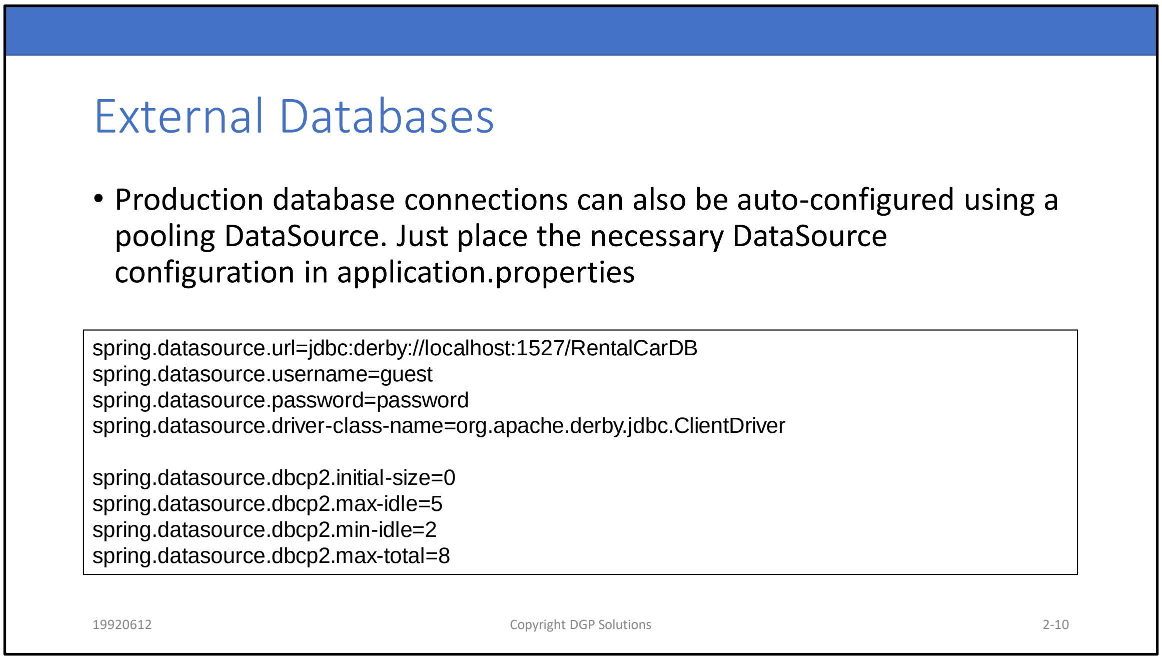
```
CREATE TABLE TOWN
(
  DISTRICTID    BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY,
  PHONECODE       BIGINT NOT NULL,
  DISTRICT       VARCHAR(20) NOT NULL,
  CITY          VARCHAR(20) NOT NULL,
  COUNTY         VARCHAR(20) NOT NULL,

  CONSTRAINT  PK_PLAN  PRIMARY KEY(DISTRICTID)
);
```

19920612 Copyright DGP Solutions 2-9

- Without the schema.sql defined, you will get a org.h2.jdbc.JdbcSQLException exception stating that any SQL statement you make in your code cannot be linked to a table in the embedded H2 database

```
INSERT INTO Employee VALUES(DEFAULT,'Fred','Flintstone',200);
INSERT INTO Employee VALUES(DEFAULT,'Eric','Colbert',300);
INSERT INTO Employee VALUES(DEFAULT,'Mary','Contrary',200);
INSERT INTO Employee VALUES(DEFAULT,'Jane','Doe',100);
INSERT INTO Employee VALUES(DEFAULT,'Tony','Lancealot',300);
INSERT INTO Employee VALUES(DEFAULT,'Anna','Pittstop',100);
```



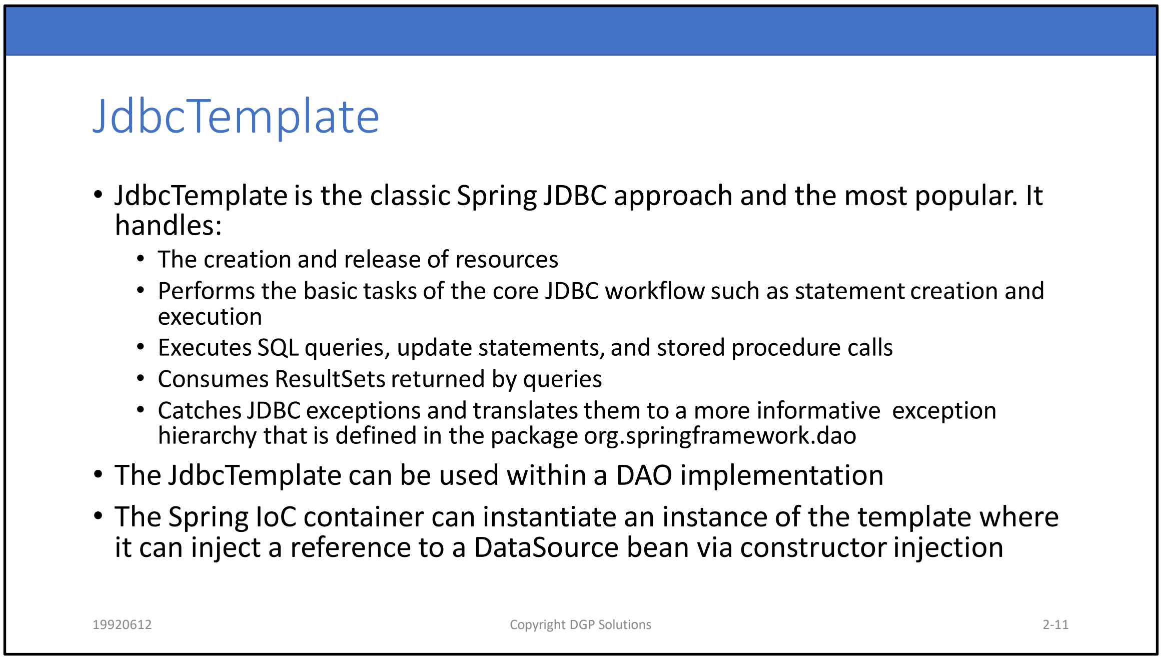
External Databases

- Production database connections can also be auto-configured using a pooling DataSource. Just place the necessary DataSource configuration in application.properties

```
spring.datasource.url=jdbc:derby://localhost:1527/RentalCarDB
spring.datasource.username=guest
spring.datasource.password=password
spring.datasource.driver-class-name=org.apache.derby.jdbc.ClientDriver

spring.datasource.dbcp2.initial-size=0
spring.datasource.dbcp2.max-idle=5
spring.datasource.dbcp2.min-idle=2
spring.datasource.dbcp2.max-total=8
```

```
<dependency>
<groupId>org.apache.derby</groupId>
<artifactId>derbyclient</artifactId>
<version>10.14.5.0</version>
</dependency>
```



JdbcTemplate

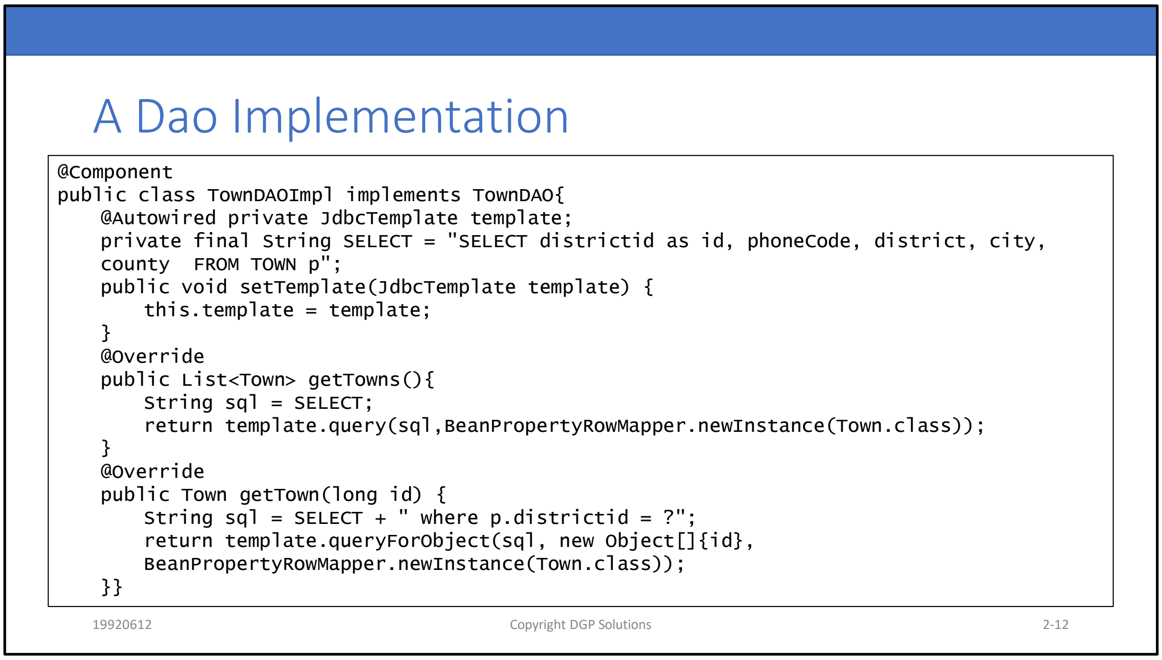
- JdbcTemplate is the classic Spring JDBC approach and the most popular. It handles:
  - The creation and release of resources
  - Performs the basic tasks of the core JDBC workflow such as statement creation and execution
  - Executes SQL queries, update statements, and stored procedure calls
  - Consumes ResultSets returned by queries
  - Catches JDBC exceptions and translates them to a more informative exception hierarchy that is defined in the package org.springframework.dao
- The JdbcTemplate can be used within a DAO implementation
- The Spring IoC container can instantiate an instance of the template where it can inject a reference to a DataSource bean via constructor injection

19920612 Copyright DGP Solutions 2-11

- Shown below is a public JdbcTemplate(DataSource dataSource) in a Java configuration bean

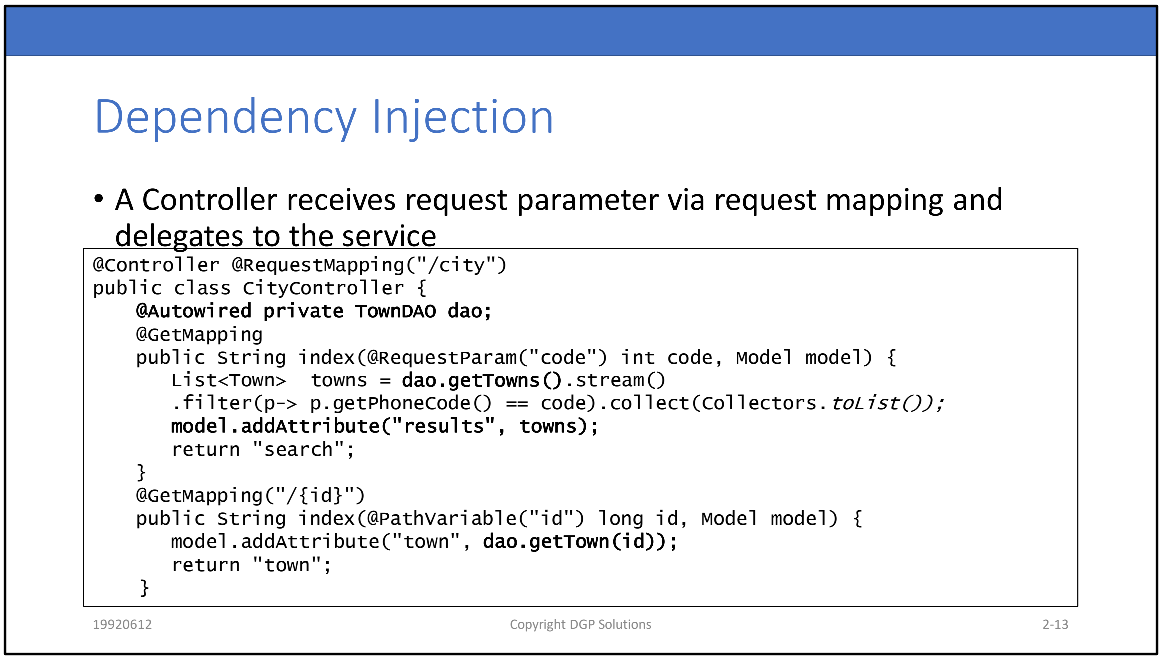
```
@Autowired
private DataSource ds;

@Bean
JdbcTemplate getJdbcTemplate() {
  return new JdbcTemplate(ds);
}
```



A Dao Implementation

```
@Component
public class TownDAOImpl implements TownDAO{
    @Autowired private JdbcTemplate template;
    private final String SELECT = "SELECT districtid as id, phoneCode, district, city,
    county  FROM TOWN p";
    public void setTemplate(JdbcTemplate template) {
        this.template = template;
    }
    @Override
    public List<Town> getTowns(){
        String sql = SELECT;
        return template.query(sql,BeanPropertyRowMapper.newInstance(Town.class));
    }
    @Override
    public Town getTown(long id) {
        String sql = SELECT + " where p.districtid = ?";
        return template.queryForObject(sql, new Object[]{id},
        BeanPropertyRowMapper.newInstance(Town.class));
    }}
```

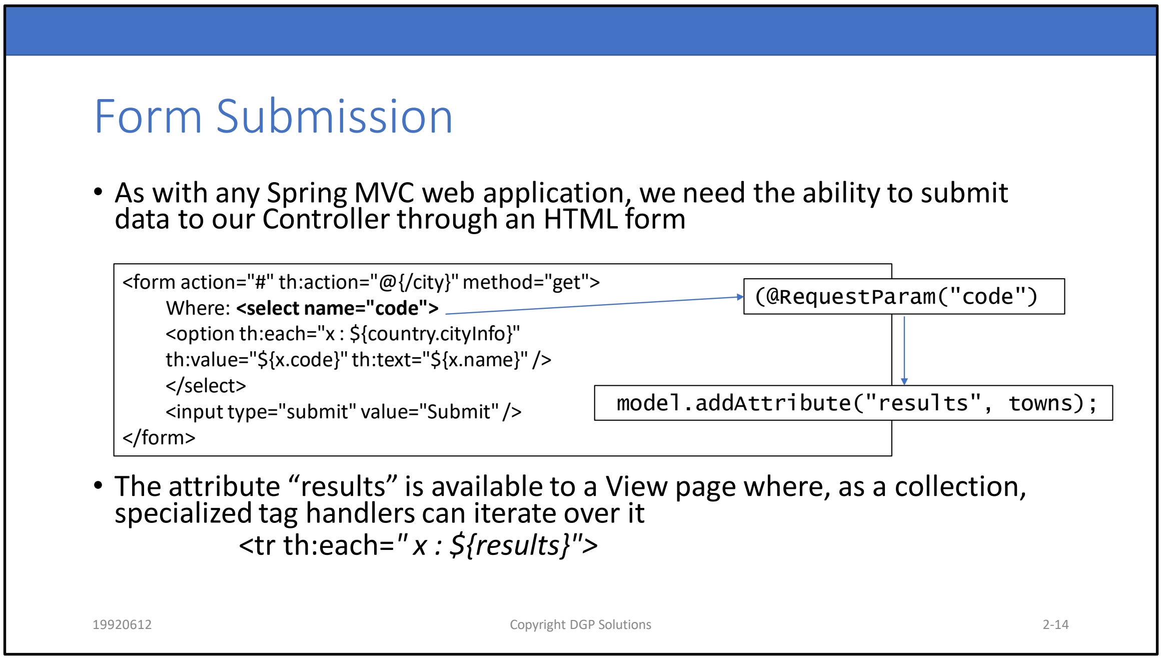


Dependency Injection

- A Controller receives request parameter via request mapping and delegates to the service

```
@Controller @RequestMapping("/city")
public class CityController {
    @Autowired private TownDAO dao;
    @GetMapping
    public String index(@RequestParam("code") int code, Model model) {
       List<Town>  towns = dao.getTowns().stream()
       .filter(p-> p.getPhoneCode() == code).collect(Collectors.toList());
       model.addAttribute("results", towns);
       return "search";
    }
    @GetMapping("/{id}")
    public String index(@PathVariable("id") long id, Model model) {
       model.addAttribute("town", dao.getTown(id));
       return "town";
    }
```

19920612 Copyright DGP Solutions 2-13



Form Submission

- As with any Spring MVC web application, we need the ability to submit data to our Controller through an HTML form

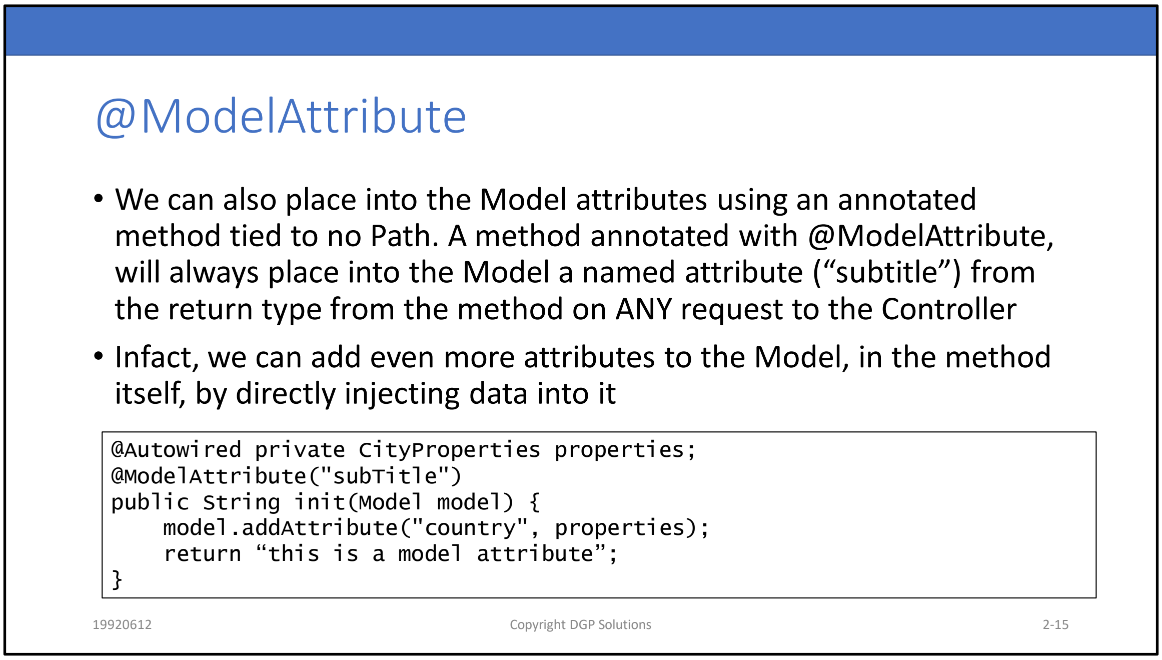
```
<form action="#" th:action="@{/city}" method="get">
    Where: <select name="code">
    <option th:each="x : ${country.cityInfo}"
    th:value="${x.code}" th:text="${x.name}" />
    </select>
    <input type="submit" value="Submit" />
</form>
```

```
(@RequestParam("code")
```

```
model.addAttribute("results", towns);
```

- The attribute “results” is available to a View page where, as a collection, specialized tag handlers can iterate over it

  <tr th:each="*x : ${results}*">



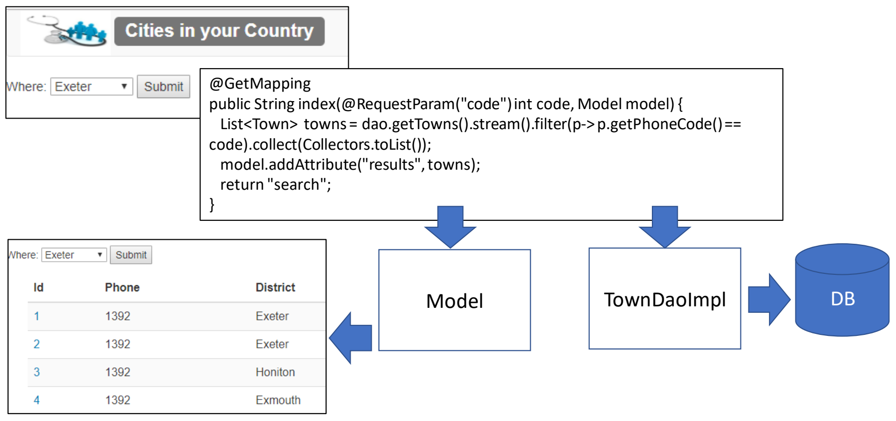
@ModelAttribute

- We can also place into the Model attributes using an annotated method tied to no Path. A method annotated with @ModelAttribute, will always place into the Model a named attribute (“subtitle”) from the return type from the method on ANY request to the Controller
- Infact, we can add even more attributes to the Model, in the method itself, by directly injecting data into it

```
@Autowired private CityProperties properties;
@ModelAttribute("subTitle")
public String init(Model model) {
    model.addAttribute("country", properties);
    return “this is a model attribute”;
}
```

19920612 Copyright DGP Solutions 2-15

Invoking our Controller



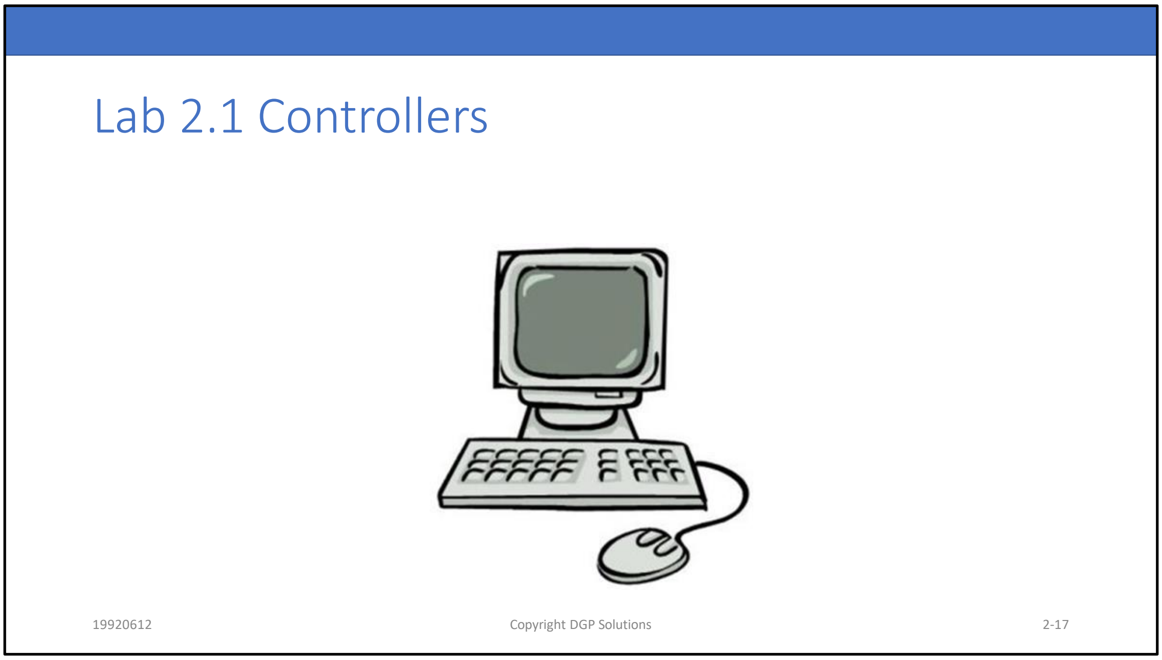
Cities in your Country

Where: Exeter Submit

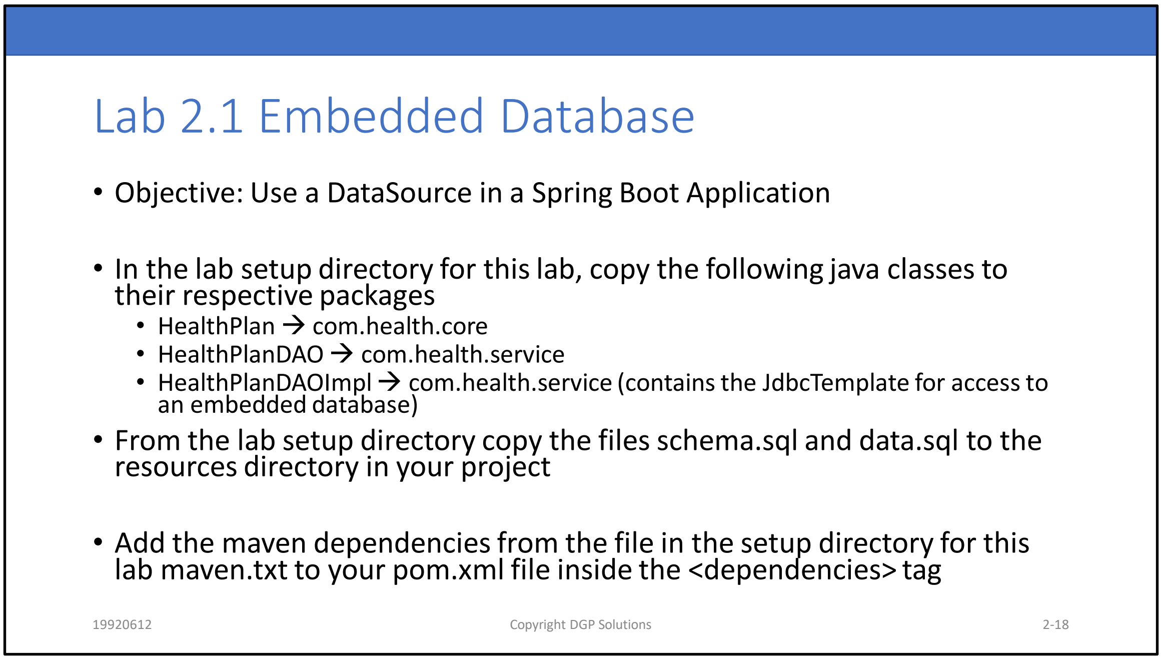
```
@GetMapping
public String index(@RequestParam("code") int code, Model model) {
  List<Town> towns = dao.getTowns().stream().filter(p-> p.getPhoneCode() ==
code).collect(Collectors.toList());
  model.addAttribute("results", towns);
  return "search";
}
```

Where: Exeter Submit

Id	Phone	District
1	1392	Exeter
2	1392	Exeter
3	1392	Honiton
4	1392	Exmouth



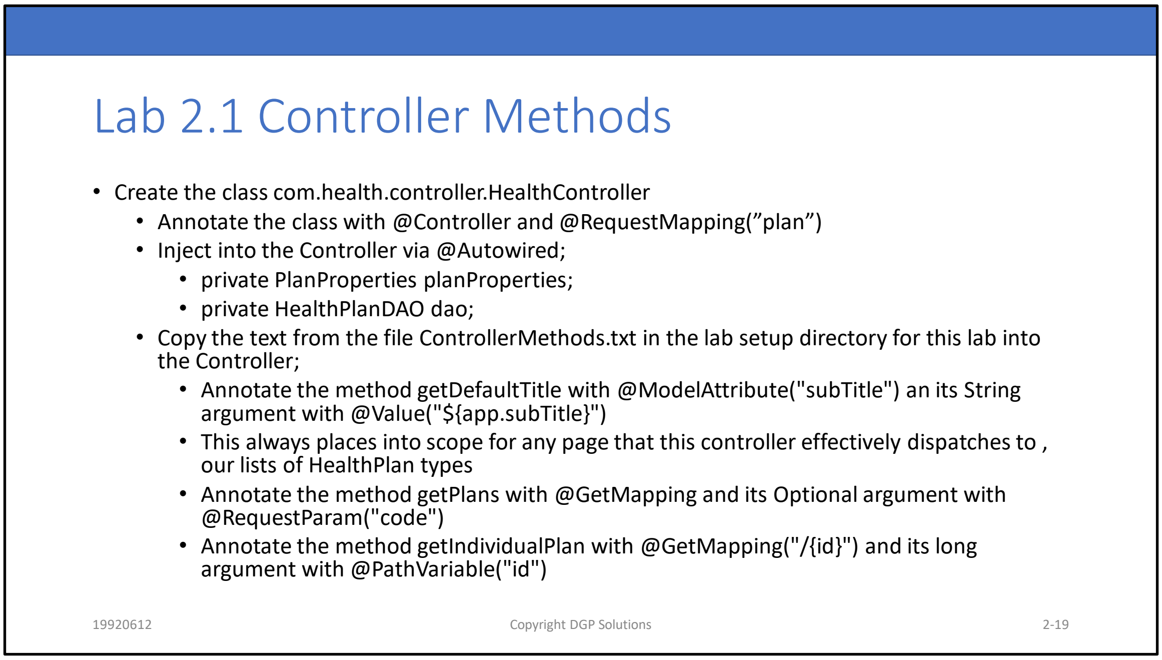
Lab 2.1 Controllers



Lab 2.1 Embedded Database

- Objective: Use a DataSource in a Spring Boot Application
- In the lab setup directory for this lab, copy the following java classes to their respective packages
  - HealthPlan → com.health.core
  - HealthPlanDAO → com.health.service
  - HealthPlanDAOImpl → com.health.service (contains the JdbcTemplate for access to an embedded database)
- From the lab setup directory copy the files schema.sql and data.sql to the resources directory in your project
- Add the maven dependencies from the file in the setup directory for this lab maven.txt to your pom.xml file inside the <dependencies> tag

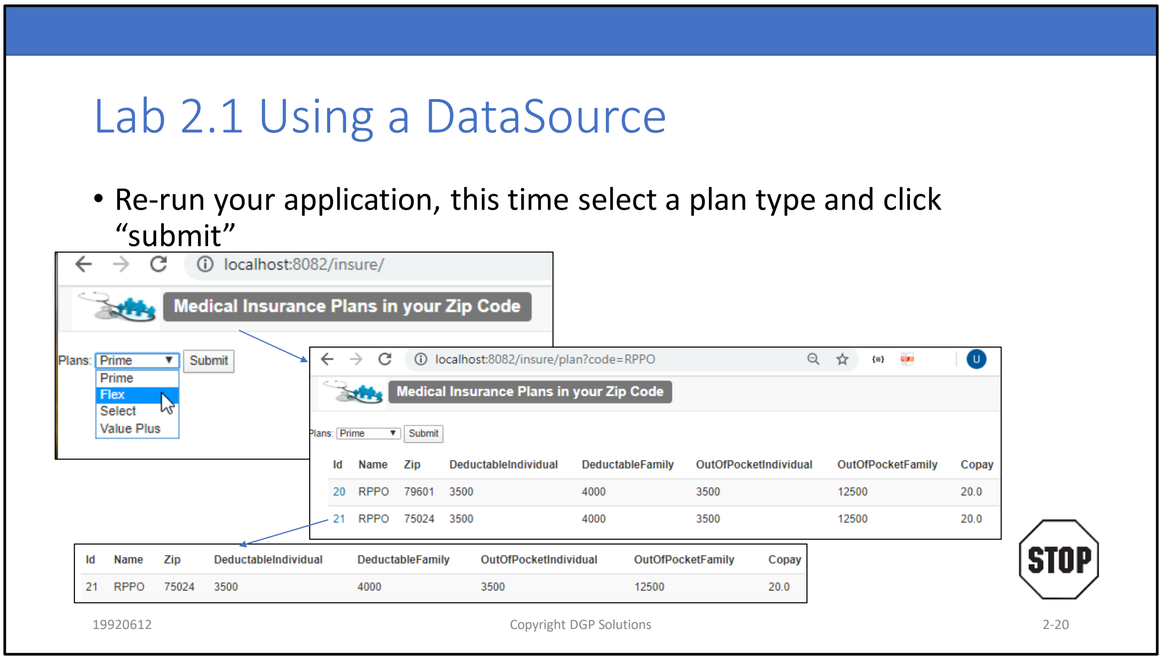
19920612 Copyright DGP Solutions 2-18



Lab 2.1 Controller Methods

- Create the class com.health.controller.HealthController
  - Annotate the class with @Controller and @RequestMapping(”plan”)
  - Inject into the Controller via @Autowired;
    - private PlanProperties planProperties;
    - private HealthPlanDAO dao;
  - Copy the text from the file ControllerMethods.txt in the lab setup directory for this lab into the Controller;
    - Annotate the method getDefaultTitle with @ModelAttribute("subTitle") an its String argument with @Value("${app.subTitle}")
    - This always places into scope for any page that this controller effectively dispatches to , our lists of HealthPlan types
    - Annotate the method getPlans with @GetMapping and its Optional argument with @RequestParam("code")
    - Annotate the method getIndividualPlan with @GetMapping("/{id}") and its long argument with @PathVariable("id")

19920612 Copyright DGP Solutions 2-19



Lab 2.1 Using a DataSource

- Re-run your application, this time select a plan type and click "submit"

localhost:8082/insure/

Medical Insurance Plans in your Zip Code

Plans: Prime ▼ Submit
Prime
Flex
Select
Value Plus

localhost:8082/insure/plan?code=RPPO

Medical Insurance Plans in your Zip Code

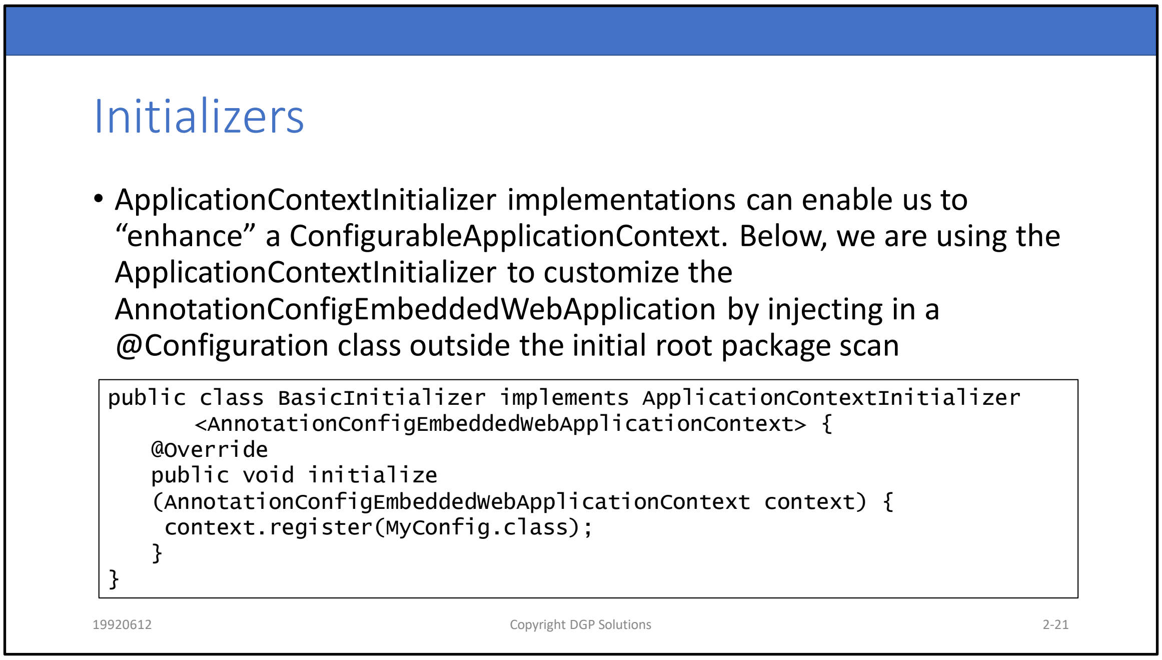
Plans: Prime ▼ Submit

Id	Name	Zip	DeductableIndividual	DeductableFamily	OutOfPocketIndividual	OutOfPocketFamily	Copay
20	RPPO	79601	3500	4000	3500	12500	20.0
21	RPPO	75024	3500	4000	3500	12500	20.0

Id	Name	Zip	DeductableIndividual	DeductableFamily	OutOfPocketIndividual	OutOfPocketFamily	Copay
21	RPPO	75024	3500	4000	3500	12500	20.0

19920612 Copyright DGP Solutions 2-20

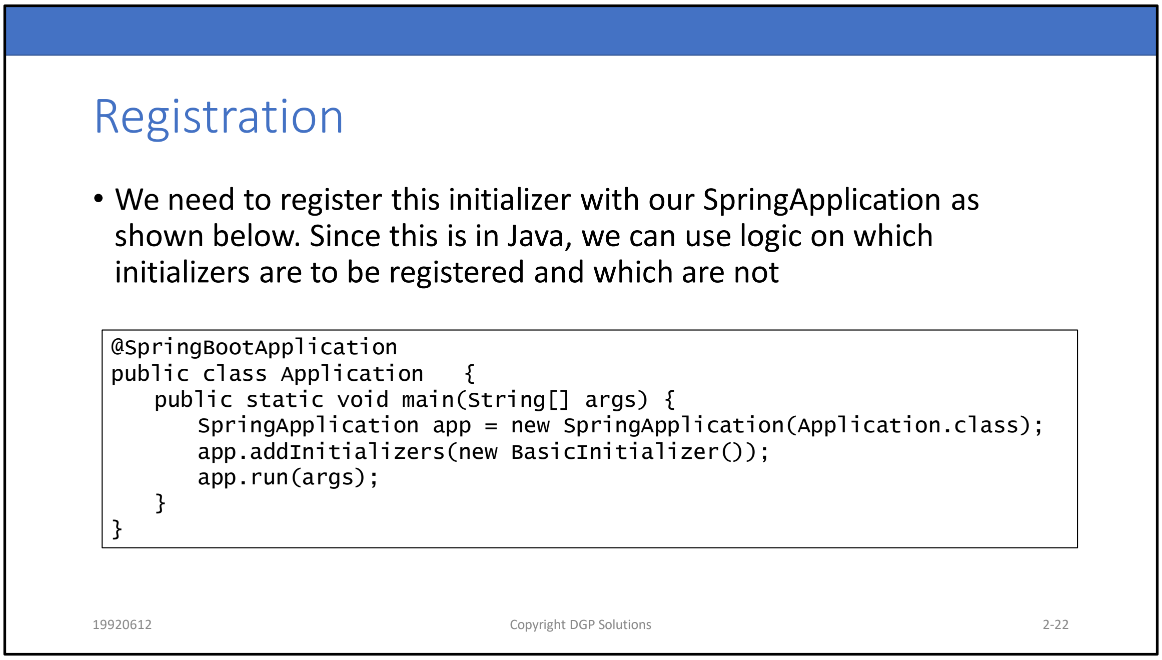
- Test the Optional with http://localhost:8082/insure/plan



Initializers

- ApplicationContextInitializer implementations can enable us to “enhance” a ConfigurableApplicationContext. Below, we are using the ApplicationContextInitializer to customize the AnnotationConfigEmbeddedWebApplication by injecting in a @Configuration class outside the initial root package scan

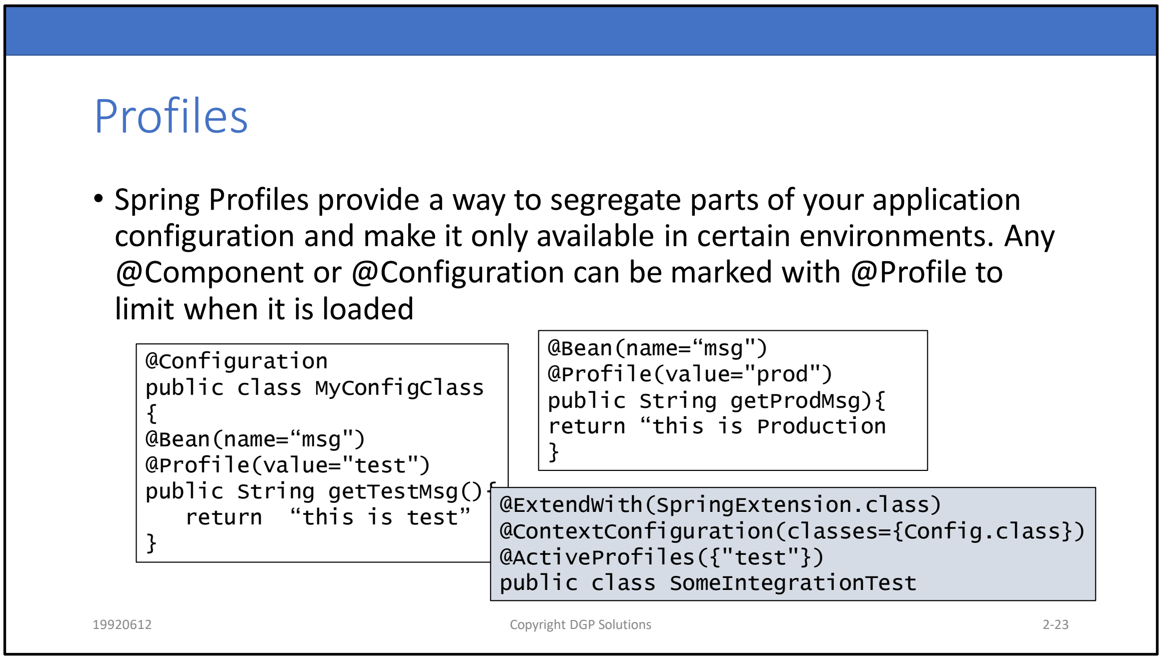
```
public class BasicInitializer implements ApplicationContextInitializer
      <AnnotationConfigEmbeddedWebApplicationContext> {
   @Override
   public void initialize
   (AnnotationConfigEmbeddedWebApplicationContext context) {
    context.register(MyConfig.class);
   }
}
```



Registration

- We need to register this initializer with our SpringApplication as shown below. Since this is in Java, we can use logic on which initializers are to be registered and which are not

```
@SpringBootApplication
public class Application   {
   public static void main(String[] args) {
      SpringApplication app = new SpringApplication(Application.class);
      app.addInitializers(new BasicInitializer());
      app.run(args);
   }
}
```



Profiles

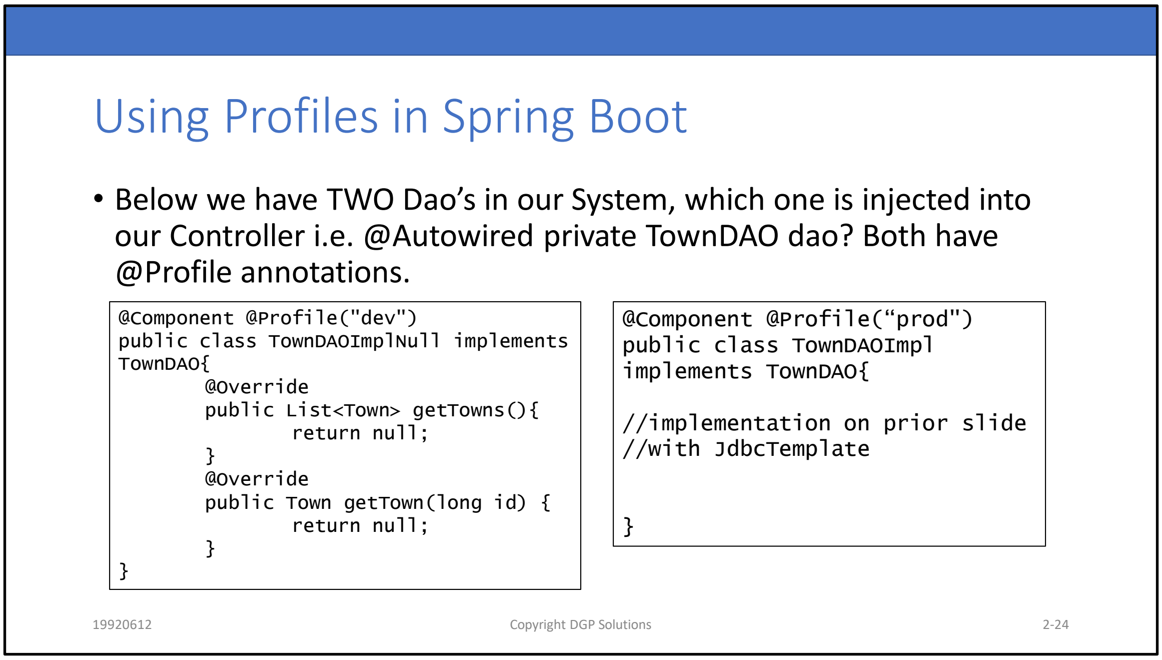
- Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments. Any @Component or @Configuration can be marked with @Profile to limit when it is loaded

```
@Configuration
public class MyConfigClass
{
@Bean(name=“msg")
@Profile(value="test")
public String getTestMsg(){
   return  “this is test”
}
```

```
@Bean(name=“msg")
@Profile(value="prod")
public String getProdMsg){
return “this is Production
}
```

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes={Config.class})
@ActiveProfiles({"test"})
public class SomeIntegrationTest
```

19920612 Copyright DGP Solutions 2-23



Using Profiles in Spring Boot

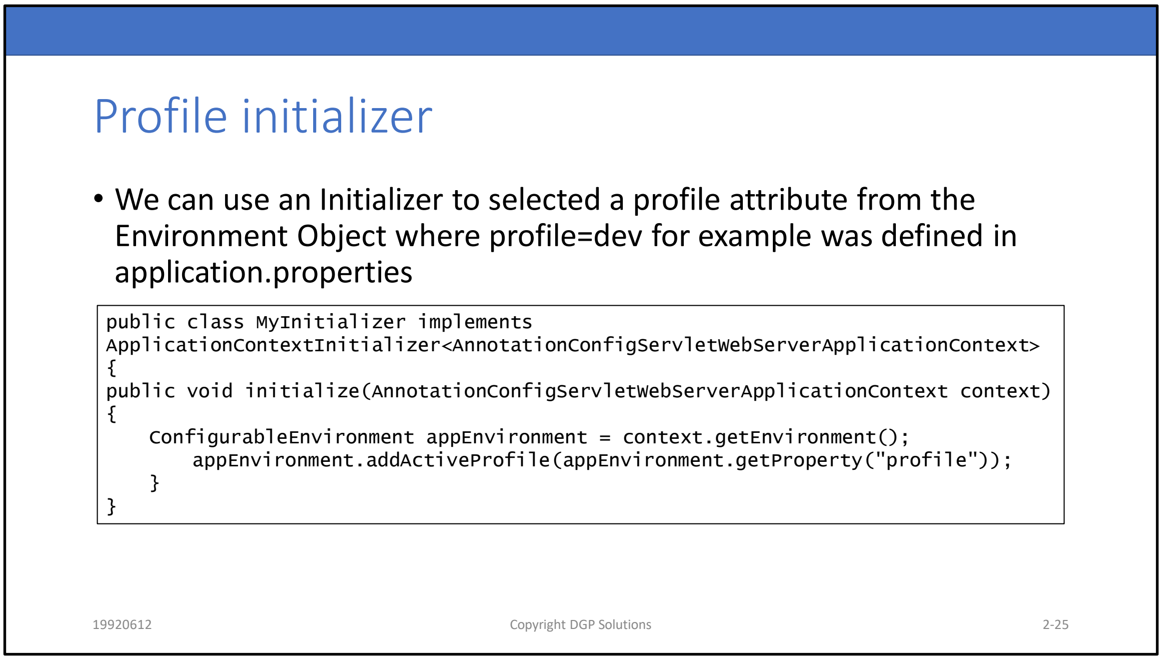
- Below we have TWO Dao’s in our System, which one is injected into our Controller i.e. @Autowired private TownDAO dao? Both have @Profile annotations.

```
@Component @Profile("dev")
public class TownDAOImplNull implements
TownDAO{
        @Override
        public List<Town> getTowns(){
                return null;
        }
        @Override
        public Town getTown(long id) {
                return null;
        }
}
```

```
@Component @Profile(“prod")
public class TownDAOImpl
implements TownDAO{

//implementation on prior slide
//with JdbcTemplate


}
```



Profile initializer

- We can use an Initializer to selected a profile attribute from the Environment Object where profile=dev for example was defined in application.properties

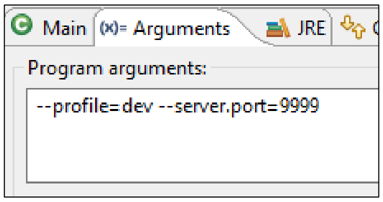
```
public class MyInitializer implements
ApplicationContextInitializer<AnnotationConfigServletWebServerApplicationContext>
{
public void initialize(AnnotationConfigServletWebServerApplicationContext context)
{
    ConfigurableEnvironment appEnvironment = context.getEnvironment();
        appEnvironment.addActiveProfile(appEnvironment.getProperty("profile"));
    }
}
```

19920612 Copyright DGP Solutions 2-25

- We can set the profile in the Environment object by a simple entry in application properties
    - spring.profiles.active=dev
- However, we could make this dynamic by using a command line argument of, profile=dev (see next slide)

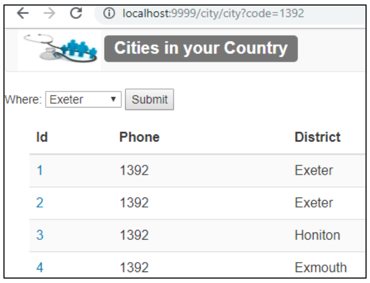
Command Line arguments

- By default, the Application class will convert any command line option argument (starting with ‘--’, e.g. --server.port=9999) to a property and add it to the Spring Environment. The, command line properties always take precedence over other property sources such as application.properties



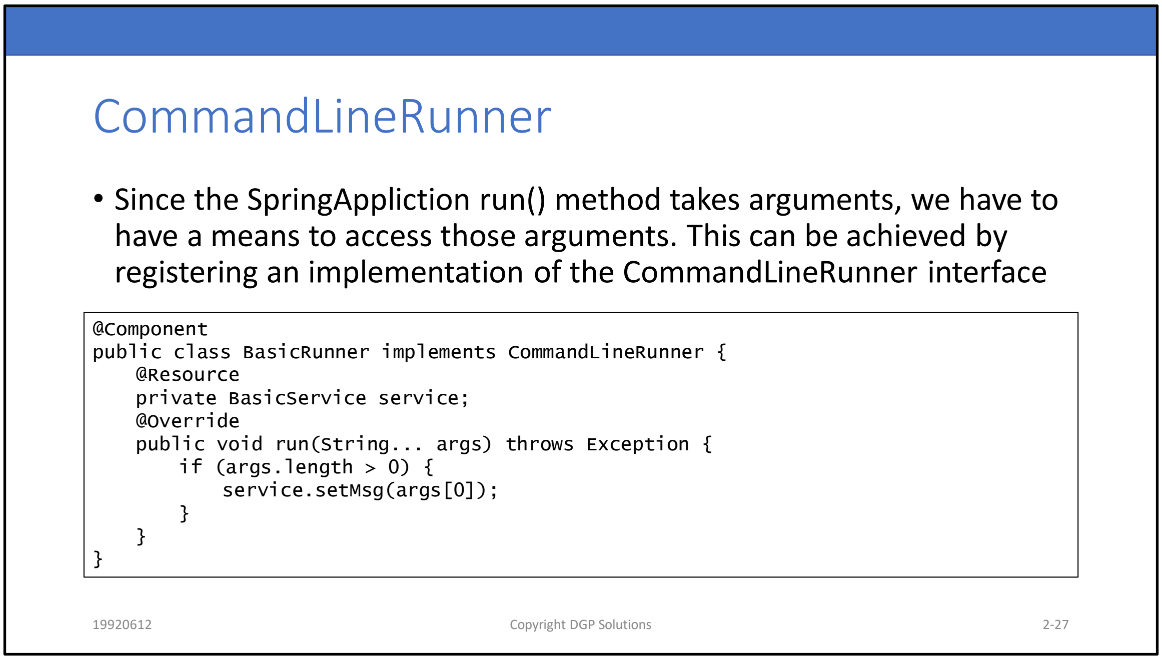
Program arguments:

```
--profile=dev --server.port=9999
```



Where: Exeter

Id	Phone	District
1	1392	Exeter
2	1392	Exeter
3	1392	Honiton
4	1392	Exmouth

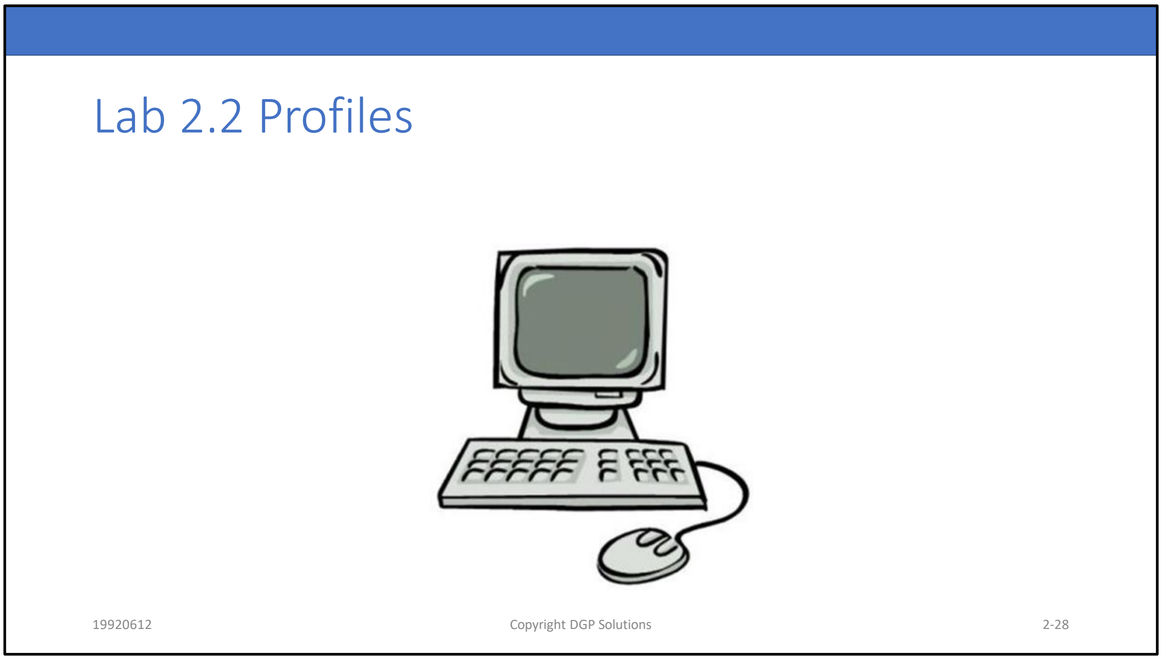


CommandLineRunner

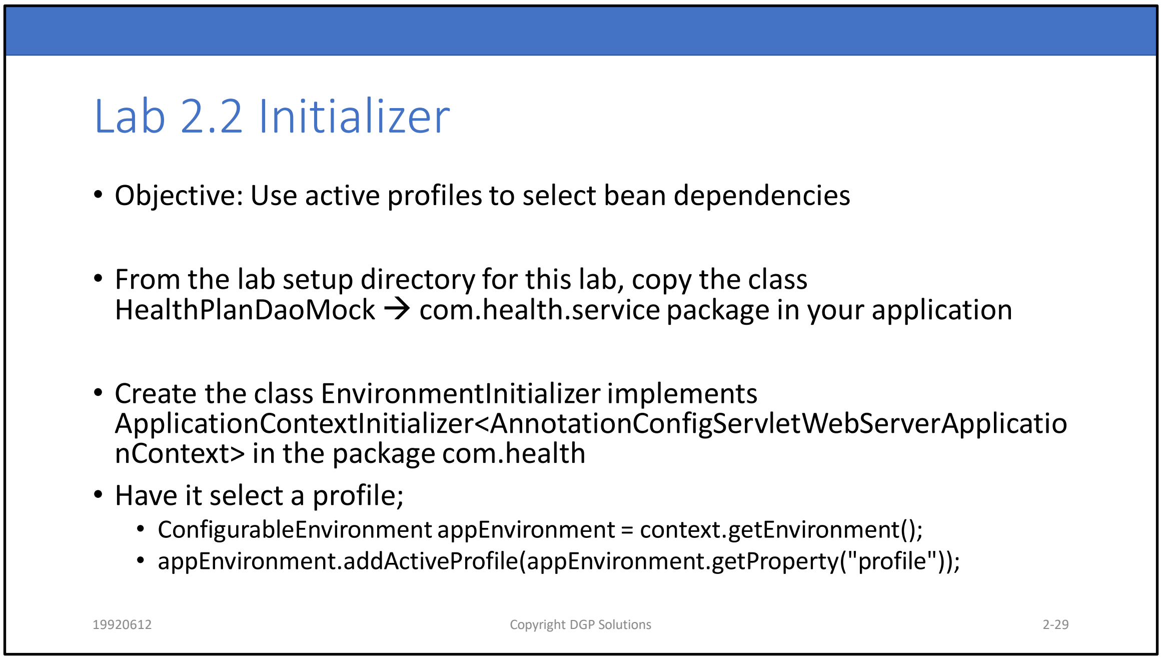
- Since the SpringAppliction run() method takes arguments, we have to have a means to access those arguments. This can be achieved by registering an implementation of the CommandLineRunner interface

```
@Component
public class BasicRunner implements CommandLineRunner {
    @Resource
    private BasicService service;
    @Override
    public void run(String... args) throws Exception {
        if (args.length > 0) {
            service.setMsg(args[0]);
        }
    }
}
```

19920612 Copyright DGP Solutions 2-27

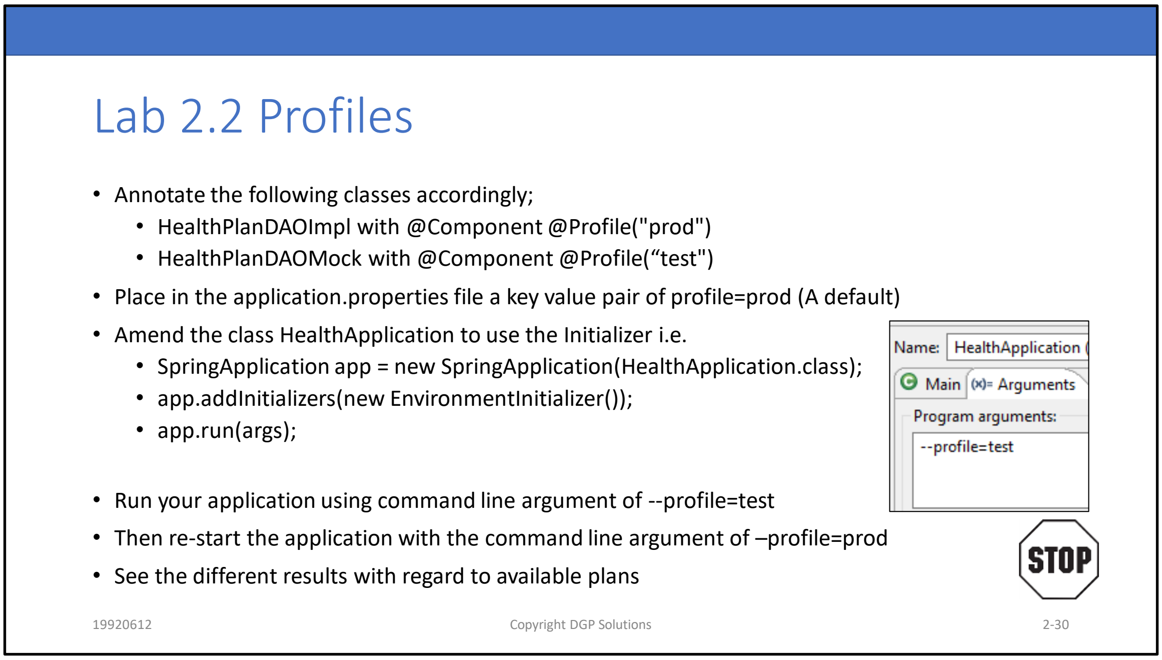


Lab 2.2 Profiles



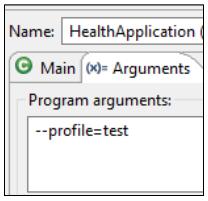
Lab 2.2 Initializer

- Objective: Use active profiles to select bean dependencies
- From the lab setup directory for this lab, copy the class HealthPlanDaoMock → com.health.service package in your application
- Create the class EnvironmentInitializer implements ApplicationContextInitializer<AnnotationConfigServletWebServerApplicationContext> in the package com.health
- Have it select a profile;
  - ConfigurableEnvironment appEnvironment = context.getEnvironment();
  - appEnvironment.addActiveProfile(appEnvironment.getProperty("profile"));



Lab 2.2 Profiles

- Annotate the following classes accordingly;
  - HealthPlanDAOImpl with @Component @Profile("prod")
  - HealthPlanDAOMock with @Component @Profile(“test")
- Place in the application.properties file a key value pair of profile=prod (A default)
- Amend the class HealthApplication to use the Initializer i.e.
  - SpringApplication app = new SpringApplication(HealthApplication.class);
  - app.addInitializers(new EnvironmentInitializer());
  - app.run(args);



Name: HealthApplication

Program arguments:

```
--profile=test
```

- Run your application using command line argument of --profile=test
- Then re-start the application with the command line argument of –profile=prod
- See the different results with regard to available plans