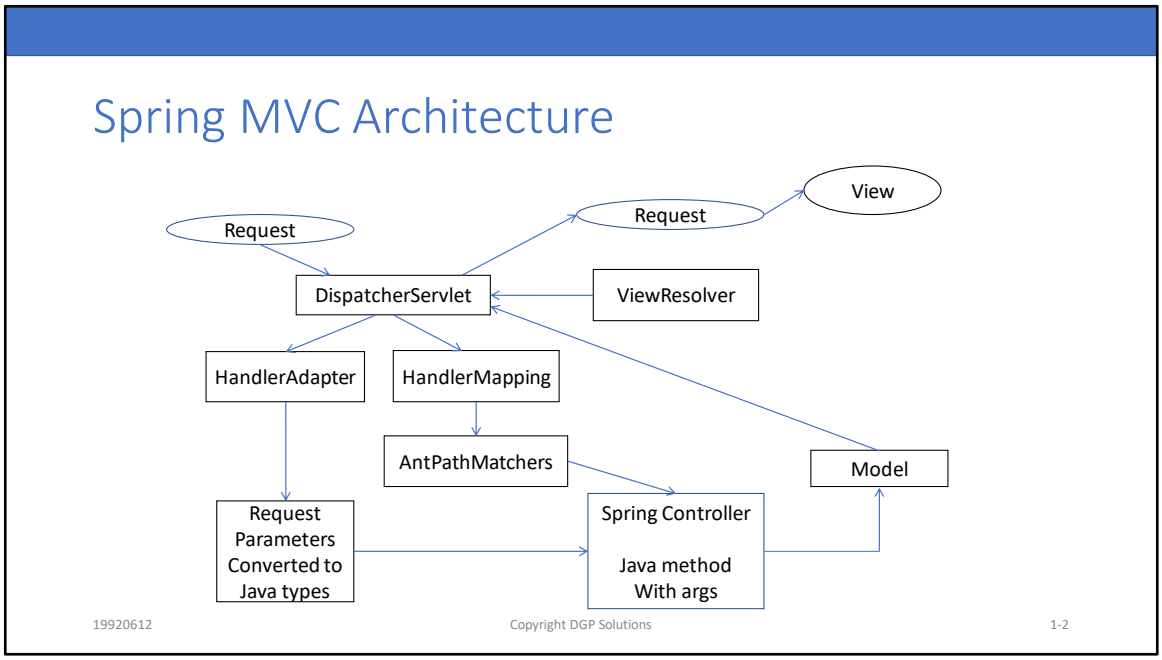
1. Spring Boot Configuration

DGP Solutions



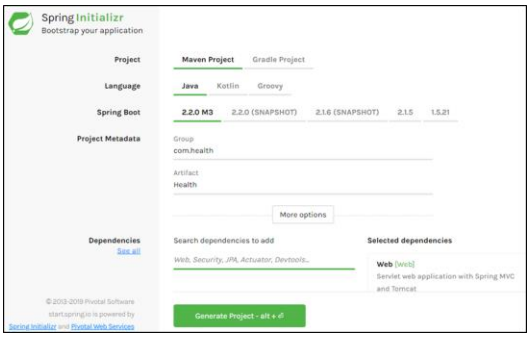
- A request is received by a Web ApplicationServer which extracts the textual request into an HttpServletRequest Object containing request Parameters and an HttpResponseObject
- The Front Controller DispatcherServlet receives all requests through a wild carded Url. Via Spring configuration, it identifies the Controller method to delegate the request to via a series of HandlerMappings and AntPathMatchers that match Url patterns to Controller Methods
- Once the method is identified, if it takes Java Typed arguments, the corresponding request parameters are converted into their respective Java Types via the registered HandlerAdapter
- The Controller method is executed and interfaces with Spring's org.springframework.ui.Model object. It can return data that is to be stored in the Model object as attributes. It can also return a View object that is also stored in the Model
- The Model is returned to the DispatcherServlet where the Model's attributes are placed into the HttpServletRequest
- The View in the model is passed to a ViewResolver to determine what View is to be dispatched to. The DispatcherServlet then dispatches to the desired View where upon the Web container renders the View Response

What is Spring Boot?

- Spring Boot makes it easy to create stand-alone Spring based Applications. It takes an opinionated view of Spring with a multitude of defaults so that you do not need to concern yourself with the details of Spring MVC and therefore, you can get an application up and running in very little time
- The primary goals of the Spring Boot project are:
  - Provide a quick means to getting started
  - Be opinionated out of the box, using provided defaults that can create a Spring MVC Web Application
  - Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration)
  - Require no code generation and no requirement for XML configuration

https://start.spring.io/

- Spring Boot provides a website that is used to create new Spring Boot projects at https://start.spring.io/



Spring Boot Starters

- Spring Boot Starter Maven dependencies simply dependency management.
- Instead of manually specifying the dependencies that your want, just add Spring Boot starter dependencies
- These maven entries are virtual packages that pull in other dependencies while containing no code of their own

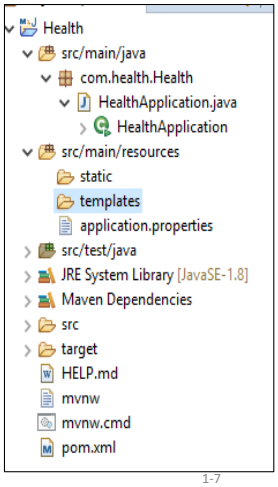
Application Class

- Our Application class will be responsible for launching our application. The @SpringBootApplication encapsulates three other annotations: @Configuration, @EnableAutoConfiguration, and @ComponentScan. Spring Managed Beans, such as Controllers, will be scanned for from the current base package and any sub packages by default. Of course we can change this with our own @ComponentScan annotation on the class

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

Project Structure

- By default, Spring Boot will serve static content (CSS, images) from a directory called /static (or /public or /resources or /META-INF/resources) in the classpath or from the root of the ServletContext
- Using an HTML5 Thymeleaf ViewResolver will look for pages in the /templates directory
- An /error mapping by default handles all errors, and it is registered as a 'global' error page in the servlet container
- The underlying Spring MVC architecture uses the HttpMessageConverter interface to convert HTTP requests and responses such as XML and Json to and from Objects



We get an Embedded server as well

- Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers. By default an embedded Tomcat server will listen for HTTP requests on the default port of 8080
- Spring Boot uses an EmbeddedWebApplicationContext that bootstraps itself by searching for a single EmbeddedServletContainerFactory bean such as a TomcatEmbeddedServletContainerFactory, that has been auto-configured
- Although we can deploy Spring Boot applications as WARS, more often than not we create executable jar components with their own server for distribution

Our Controller

- Our Controller is located via the stereotype @Controller annotation via the SpringApplication @ComponentScan. Each Controller Method is mapped to a Url and HTTP Method. Frequently via a mapping at the class and method level as below.

```
@Controller
@RequestMapping("/")
public class HomeController {
    @GetMapping
    public String index() {
        return “home”;
    }
}
```

```
Tomcat initialized with port(s): 8080 (http)
Starting service [Tomcat]
Starting Servlet engine: [Apache Tomcat/9.0.19]
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 2605 ms
Initializing ExecutorService 'applicationTaskExecutor'
LiveReload server is running on port 35729
Tomcat started on port(s): 8080 (http) with context path ''
Started HealthApplication in 4.418 seconds (JVM running for 5.168)
```

- Invocation of the Controller’s method is achieved via browser navigation through the DispatcherServlet’s architecture
- The Controller will dispatch to a page governed by the ViewResolver i.e. templates/home.html

The Model

- Requests are mapped to Controller methods through the DispatcherServlets HandlerMapping strategy i.e.the class **org.springframework.web.servlet.handler.SimpleUrlHandlerMapping**
- Controllers also receive ona request by request basis another object called the **org.springframework.ui.Model**. It is essentially a map that a Controlelr can place key/value attirbutes
- Upon returning from the Controller to the DispatcherServlet, attributes in the Model are transferred to the request.
- Why?, because the request is available through Expression Language (EL) on the page the ViewResolver selects for The DispatcherSerlvet to dispatch to

Dependency Injection

- The controller can populate the model from Spring Managed Beans via dependency Injection

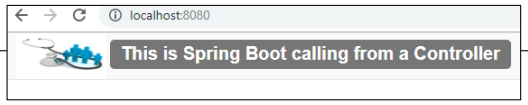
```
@Bean(name = "message")
public String getMessage() {
  return "This is Spring Boot calling from a Controller";
}
```

```
@Controller
@RequestMapping("/")
public class HomeController {
    @Autowired @Qualifier("message") private String msg;
    @GetMapping
    public String index(Model model) {
        model.addAttribute("subTitle",msg);
        return "home";
    }
}
```

Viewing the Model

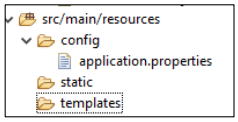
- The subsequent html picks up attributes that have now been placed into the request through EL

```
<body>
<nav class="navbar navbar-default">
<div class="container-fluid">
<div class="navbar-header">
<span><img th:src="@{/images/health.jpeg}" height="50" width="100"  /></span>
<span style="font-size:20px" th:text="${subTitle}" class="label label-
default"></span>
</div>
</div>
</nav>
</body>
```



Using application.properties

- We have an application.properties file under the src/main/resource folder directly. Frequently, it is moved to a config sub directory. We provide configurations in this file for our project such as the port and context root



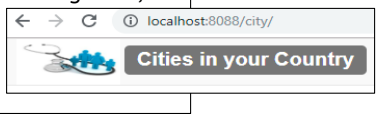

```
server.port=8088
server.servlet.context-path=/city
```

Configuration Properties

- The application.properties file can be used to hold key value pairs to provide further configuration data to your application
- These properties, are at runtime injected into the Spring Environment Object and are accessible in our Controller through Dependency Injection of the Environment object itself or the Spring @Value annotation

```
app.title=Cities
app.subTitle=${app.title} in your Country
```

```
@GetMapping
public String index(@Value("${app.subTitle}") String str,
Model model) {
   model.addAttribute("subTitle",str);
   return "home";
}
```



YAML

- An alternative to an application.properties file is to use an application.yaml file
- YAML is a superset of JSON, and as such is a convenient format within which you can specify hierarchical relationships in configuration data
- The SpringApplication class will automatically support YAML as an alternative to properties

YAML vs Properties

```
server.port=8088
server.servlet.context-path=/city

app.title=Cities
app.subTitle=${app.title} in your Country
```

```
server:
  port: 8088
  servlet:
    context-path: /city

app:
  title: Cities
  subTitle: ${app.title} in your Country
```

More Configurations

- When using the @Value("${property}") annotation to inject configuration properties into a Controller
- Although type conversions are accomplished, validation of these values is not done at application load up time. The approach is restrictive when using complex objects for our configuration properties
- Spring Boot provides an alternative method of working with properties that lets us use strongly typed beans to govern and validate the configuration of your application

@ConfigurationProperties

- The POJO Class below can act as a type safe and validated configuration bean. @ConfigurationProperties is a @Component variant. It is used to select key/value pairs from application.properties and populate an instance of the bean. Its argument restricts the transference of attributes to properties prefixed with “city” only in this case

```
@ConfigurationProperties("city")
//getter and setters not shown
public class CityProperties {
    private double areaCode;
    private List<CityInfo> cityInfo = new ArrayList<CityInfo>();
    static class CityInfo {
        private String name;
        private int code;
    }
```


Mapping properties to bean attributes

- Our application.properties file

```
city.telephone.code=50.00
city.cityInfo[0].name=Exeter
city.cityInfo[0].code=1392
city.cityInfo[1].name=Bristol
city.cityInfo[1].code=117
city.city_Info[2].NAME=Gloucester
city.cityInfo[2].code=978
```

```
@ConfigurationProperties("city")
//getter and setters not shown
public class CityProperties {
    private double areaCode;
    private List<CityInfo> cityInfo
    static class CityInfo {
        private String name;
        private int code;
    }
```

- Note how we populate the List, to form complex configurations
- Spring boot allows Relaxed binding rules i.e. almost a match is good enough

```
city.city_Info[0].NAME=Gloucester
```

JSR303 Validation

- Applying JSR303 Validations, with provided or Custom messages, ensures a valid configuration at Runtime

```
@ConfigurationProperties("city")
@Validated
public class CityProperties {
    @DecimalMax(value="50.00", message="Area Code unknown")
    private double areaCode;
    private List<CityInfo> info = new ArrayList<CityInfo>();
    static class CityInfo {
        @Size(max=5)
        private String name;
        @Max(99999)
        private int code;
        //getter and setters not shown
    }
```

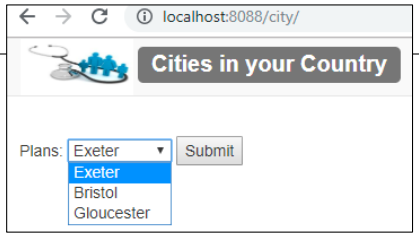
Injecting Type Safe Configuration

- By this time the configuration is compiled, validated and Strongly types

```
@Controller
@RequestMapping("/")
public class HomeController {
    @Autowired private CityProperties properties;
    @GetMapping
    public String index
    (@Value("${app.subTitle}") String str, Model model) {
       model.addAttribute("subTitle",str);
       model.addAttribute("country", properties);
       return "home";
}
}
```

Accessing Our Configuration

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



Lab 1.1 – Getting Started in Spring Boot

Lab 1.1

- **Objective** Create a basic Spring Boot Web Application with a Spring Controller using an application.yaml file for custom configurations

- Locate the startup code for this lab from your lab setup directory and import the Maven Project you will find there into your ide as an existing maven project .This project consists of a number of classes that we will eventually utilize as we build our web project through a succession of labs.

- Locate the file application.yaml. In this file, you will configure the following in yaml format to customize the Spring Boot defaults:
  - server.port=8082
  - server.servlet.context-path=/insure
  - app.title=Medical Insurance Plans
  - app.subtitle=${app.title} in your Zip Code

Lab 1.1 – Validated Configuration

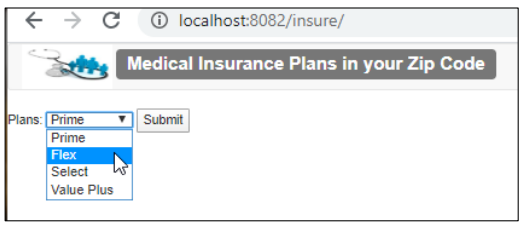
- In the class com.health.PlanProperties complete the //TODO statements as follows:
  - Annotate the class to be a @Component, @configurationProperties(“health”) and @Validated
  - Annotate the member field maxCopy with @DecimalMax(value="60.00", message="Copays are too high as it is")
  - Annotate the member field maxDeductableIndividual with @DecimalMin(value=“30.00”)
  - Annotate the PlanType member fields of symbol and name with @Size(max=7) and @Size(max=4) respectively
- Locate the file application.properties
  - Some of the properties for your configuration class have already been populated, note the relaxed binding. Add the following key value pairs
  - health.maxCopay=30
  - health.maxDeductableIndividual=6000

Lab 1.1 Controller

- In the class com.health.web.HomeController, complete the //TODO statements as follows to create a Restful Service:
  - Add a @RestController annotation to the class so it can be picked up in a scan from our Application (i.e. make it Spring managed).
  - Use @Autowired to inject in an instance of PlanProperties
  - Annotate the method public String index(), with a @GetMapping tied to the path "/". This is the URL endpoint i.e. context root.
  - Add an argument to this method of type String called "str" and a second argument of type Model called model
  - Annotate the String argument with a @Value annotation taking an expression to locate the "app.subTitle" value from your yaml file (the yaml file entry is injected into the method).
  - Add to the model two attributes;
    - model.addAttribute("subTitle",str);
    - model.addAttribute("types", planProperties);
  - Return the String "home from this method

Lab 1.1 – Controller Triggered

- In the class com.health.Application, complete the //TODO statements as follows:
  - Annotate the class with the SpringBootApplication annotation. This makes the class establish Spring Boot defaults for a Spring MVC application
  - All html pages have been provided in the templates directory
- Test your app by running the "main" method in com.health.Application. You should see in your console the Spring Boot Banner. Open a web browser to the url http://localhost:8082/insure/.



- The drop down menu is populated from your list of Plans Types that you populated from your application.properties file via the validated PlanProperties configuration class

Logging

- You have probably noticed a fair amount of logging going on in your console
- Spring Boot uses Commons logging for all internal logging but leaves the underlying log implementation open for you to use other logging frameworks such a Logback or Log4j
- In each case, loggers are pre-configured to use console output but also have optional file output available. By default, if you use the "Starter Projects", Logback is the default logging framework used.

Change Logging levels

- We can change what messages are logged via our application.properties (or YAML) file to log at different levels. Below we only log WARN and above both in our application and in the Spring Framework itself

```
logging.level.root=WARN
logging.level.org.springframework=WARN
```

- Our console is getting pretty active. Then why not log to a file? And only a file

```
logging.file=my.log
logging.pattern.console=
```


Banner File

- The banner that is printed in your console at start up can be changed by adding a banner.txt file to your classpath or by setting the spring.banner.location property to the location of such a file.
- Inside your banner.txt file, you can use placeholders to inject items from the Environment object

Welcome.txt

```
spring.application.name=healthPlan
spring.description=Provide HealthPlan quotes for Zip codes
Spring Boot Version ${spring-boot.version}
```

application.properties

```
spring.banner.location=welcome.txt
spring.banner.image.location=static/images/search.jpeg
spring.banner.image.width=20
spring.banner.image.height=10
spring.banner.image.margin=2
```

Lab 1.2 – Logging

Lab 1.2 - Logging

- Objective Use the application.properties file to manipulate logging in your Spring boot Application

- In the class HomeController;
  - Add a Logger member variable;
    - private Logger logger = LoggerFactory.getLogger(this.getClass().getSimpleName());
    - In the method index; add three logging statements
      - logger.error("This is an error message");
      - logger.warn("This is a warning message");
      - logger.info("This is an info message");
  - In the file application.properties, add the following entries;
    - logging.level.root=WARN
    - logging.level.org.springframework=INFO
    - logging.file=my.log
    - logging.pattern.console=

Lab 1.2 - Logging

- If you run your application now, logging only occurs in your mylog.log and not in the console. Additionally only some of your log statements are triggered from the HomeController.

- Change the Banner. In the file application.properties add the following key value pairs;
  - spring.banner.location=welcome.txt
  - spring.banner.image.location=static/images/search.jpeg
  - spring.banner.image.width=20
  - spring.banner.image.height=10
  - spring.banner.image.margin=2

Lab 1.2 - Banner

- Create a file Welcome.txt directly under your src/resources node. Place in it the following entries;
  - Spring Application Name=healthPlan
  - Spring Description=${app.subTitle}
  - Spring Boot Version ${spring-boot.version}
  - ${server.port}${server.servlet.context-path}
- Comment out the entry to disable console logging in your application properties
- Restart your application.

```
          o&&*
        &:.  *:o
       8 .     *:
       8  ...   *&
       8    ....o
        o    ..&
        &8o  o&
      8@&
     8@@
      #.

Spring Application Name=healthPlan
Spring Description=Medical Insurance Plans in your zip Code
Spring Boot Version 2.2.2.RELEASE
8082/insure
```