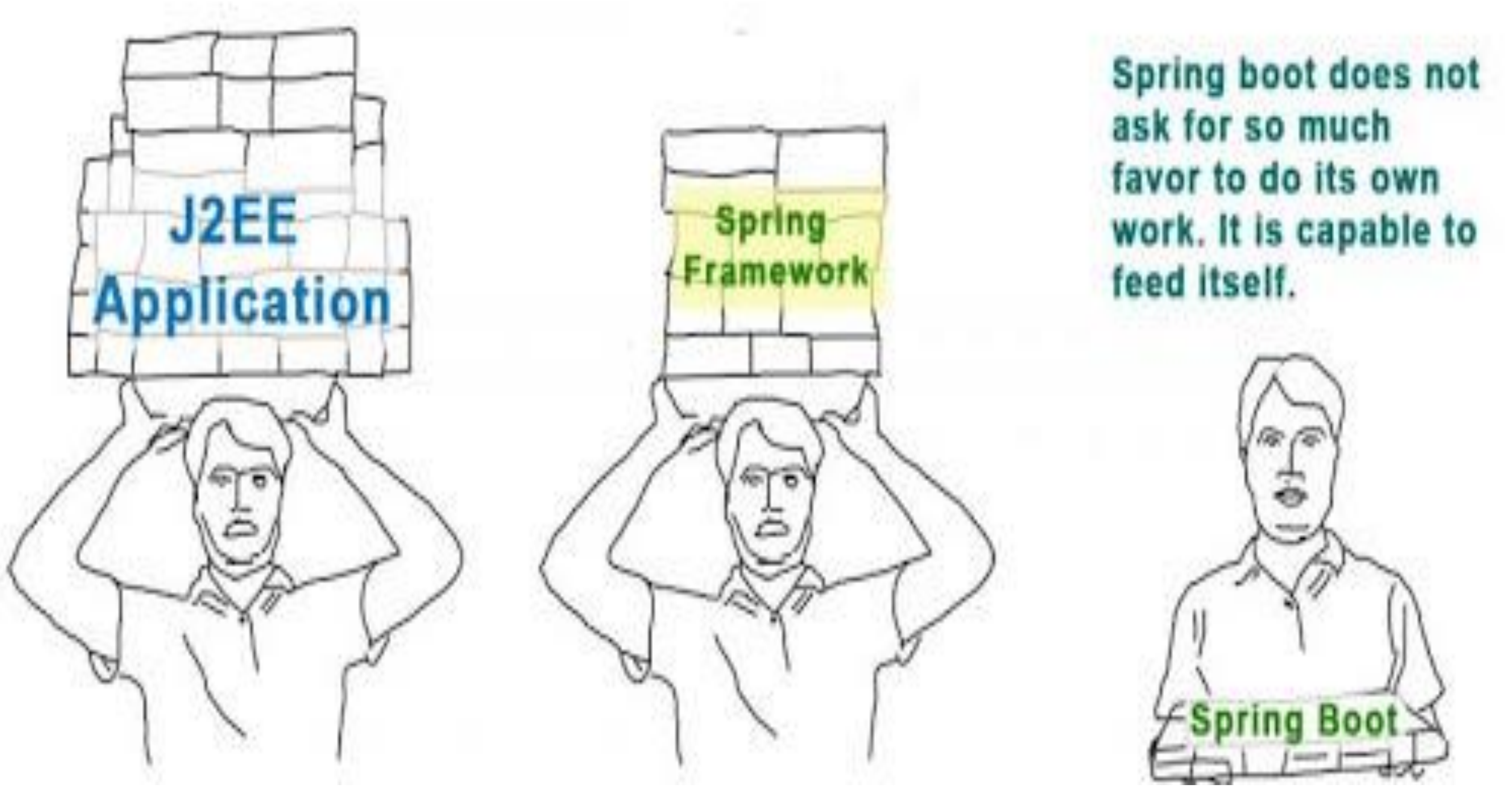
DAY 2 Spring Boot

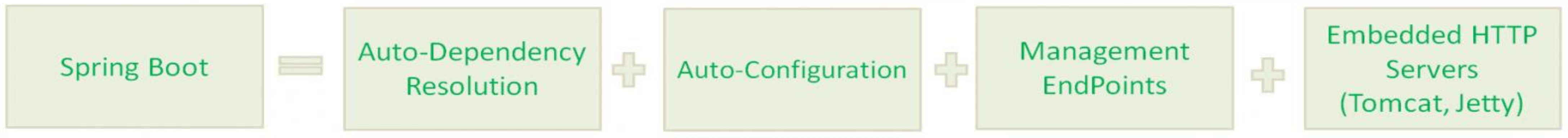
Slides By S.B.Madake


J2EE Application
Spring Framework
Spring boot does not ask for so much favor to do its own work. It is capable to feed itself.
Spring Boot

Spring Boot

What is Spring Boot?

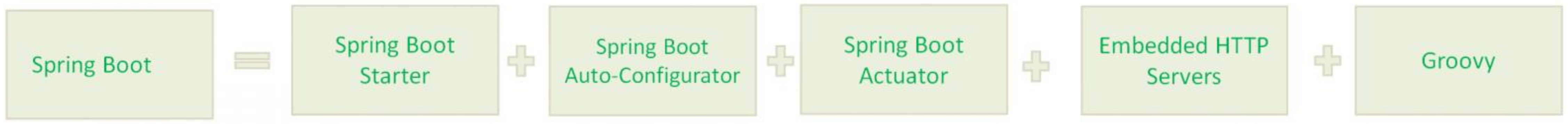
- In simple words, Spring Boot Framework is Auto-Dependency Resolution, Auto-Configuration, Management EndPoints, Embedded HTTP Servers(Jetty/Tomcat etc.) and Spring Boot CLI



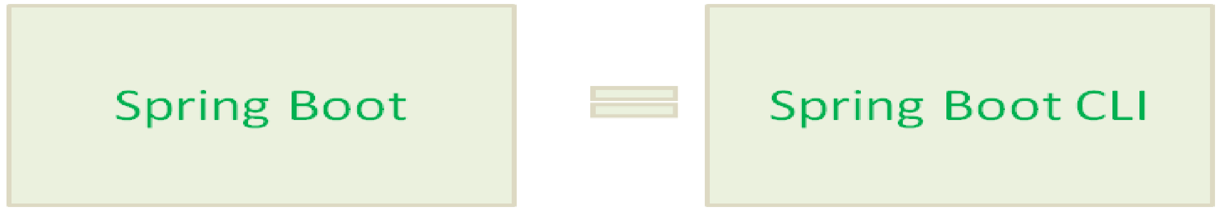
Spring Boot
Auto-Dependency Resolution
Auto-Configuration
Management EndPoints
Embedded HTTP Servers (Tomcat, Jetty)

What is Spring Boot?

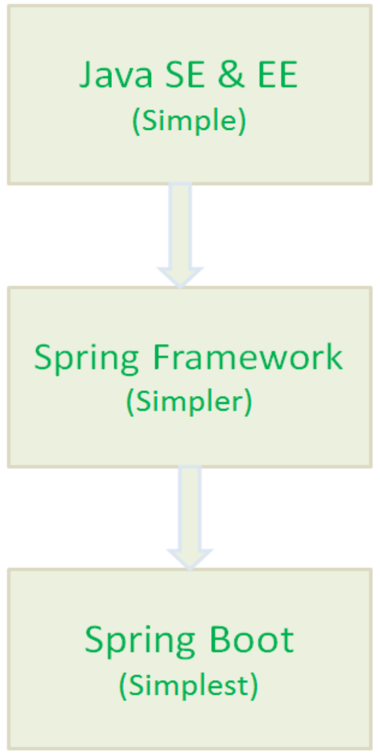
In other words, Spring Boot Framework is Spring Boot Starter, Spring Boot Auto-Configurator, Spring Boot Actuator, Embedded HTTP Servers, and Groovy.



What is Spring Boot?

- In other words, Spring Boot Framework is Spring Boot CLI.
- 

- **Why we need Spring Boot?**
- Spring Framework aims to simplify Java Applications Development.
- Spring Boot Framework aims to simplify Spring Development.



Spring Boot Components

- Spring Boot Framework has the following components:
- Spring Boot Starter
- Spring Boot AutoConfigurator
- Spring Boot Actuator
- Spring Boot CLI
- Spring Boot Initilizr

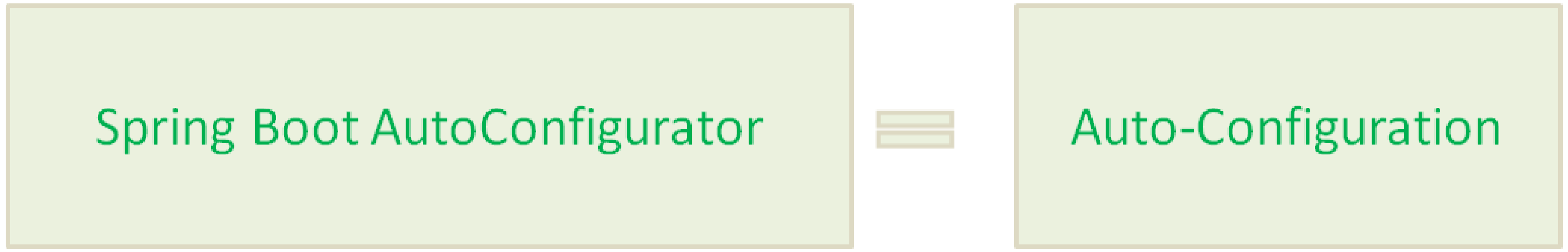
What is Spring Boot Starter?

- Spring Boot Starters are just JAR Files. They are used by Spring Boot Framework to provide “Auto-Dependency Resolution”.

Spring Boot Starter
=
Auto-Dependency Resolution

What is Spring Boot AutoConfigurator?

- Spring Boot AutoConfigurator is used by Spring Boot Framework to provide “Auto-Configuration”.



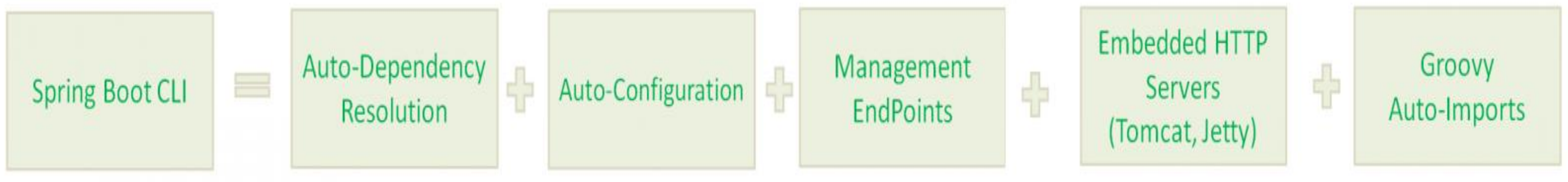
What is Spring Boot Actuator?

- Spring Boot Actuator is used by Spring Boot Framework to provide “Management EndPoints” to see Application Internals, Metrics etc.

Spring Boot Actuator = Management EndPoints

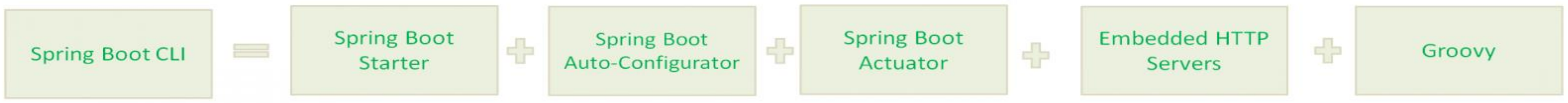
What is Spring Boot CLI?

- In simple words, Spring Boot CLI is Auto Dependency Resolution, Auto-Configuration, Management EndPoints, Embedded HTTP Servers(Jetty, Tomcat etc.) and (Groovy, Auto-Imports)



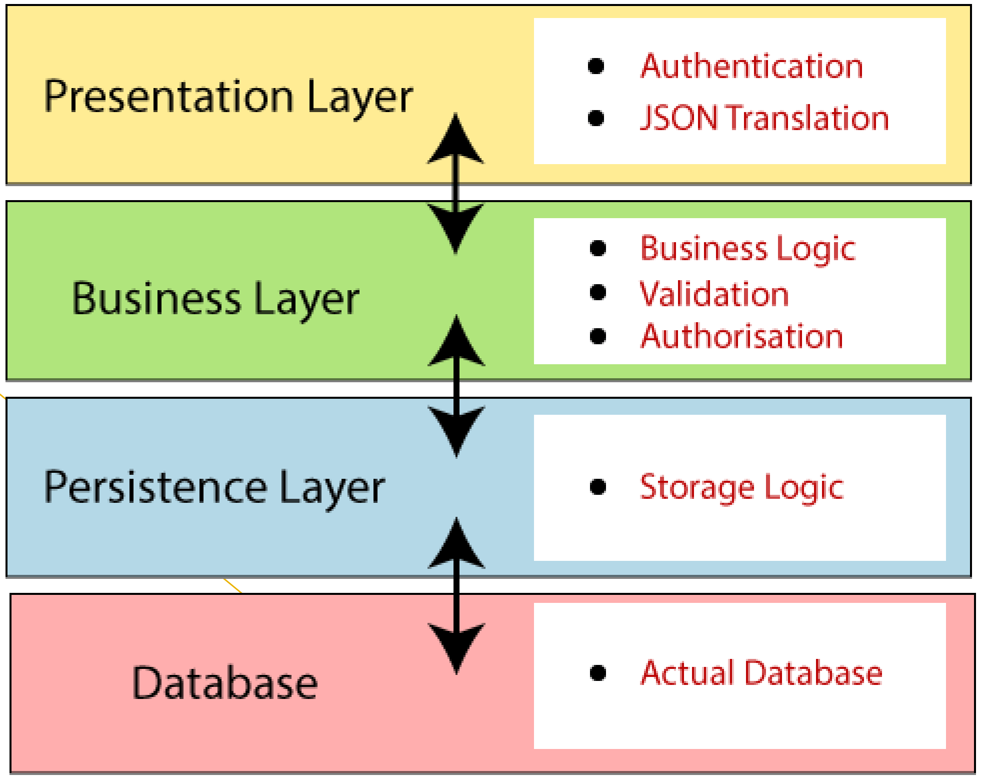
What is Spring Boot CLI?

In other words, Spring Boot CLI is Spring Boot Starter, Spring Boot Auto-Configurator, Spring Boot Actuator, Embedded HTTP Servers, and Groovy.



What is Spring Boot CLI?

Spring Boot Architecture



Presentation Layer: The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

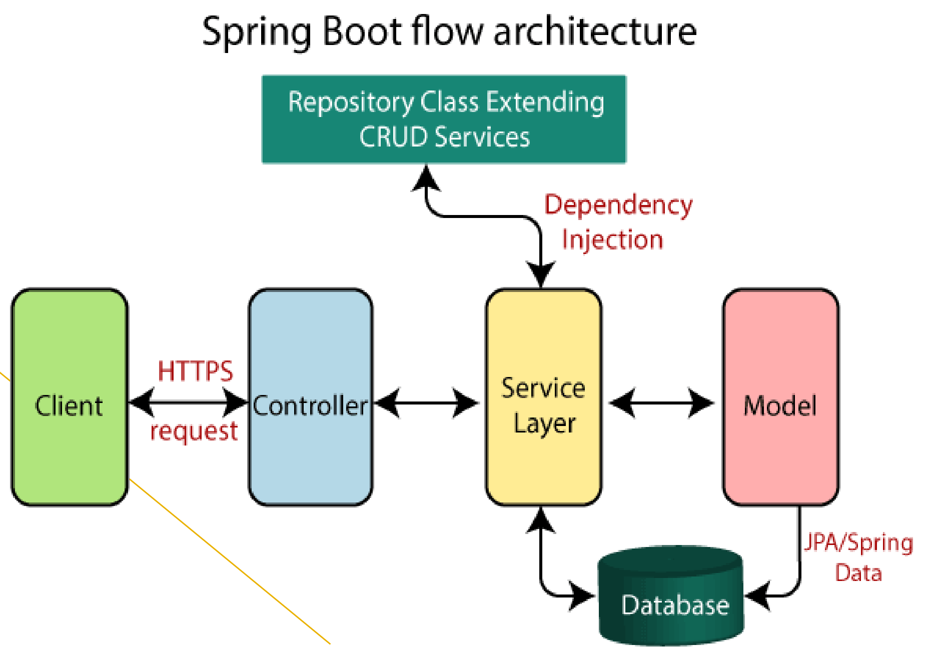
Business Layer: The business layer handles all the **business logic**. It consists of service classes and uses services provided by data access layers. It also performs **Authorization** and **validation**.

Persistence Layer: The persistence layer contains all the **storage logic** and translates business objects from and to database rows.

Database Layer: In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed

- Spring Boot is a module of the Spring Framework. It is used to create stand-alone, production-grade Spring Based Applications with minimum efforts. It is developed on top of the core Spring Framework.
- Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it.
- Before understanding the **Spring Boot Architecture**, we must know the different layers and classes present in it. There are **four** layers in Spring Boot are as follows:
- **Presentation Layer**
- **Business Layer**
- **Persistence Layer**
- **Database Layer**

Spring Boot Flow Architecture



•Now we have validator classes, view classes, and utility classes.

•Spring Boot uses all the modules of Spring-like Spring MVC, Spring Data, etc. The architecture of Spring Boot is the same as the architecture of Spring MVC, except one thing: there is no need for **DAO** and **DAOImpl** classes in Spring boot.

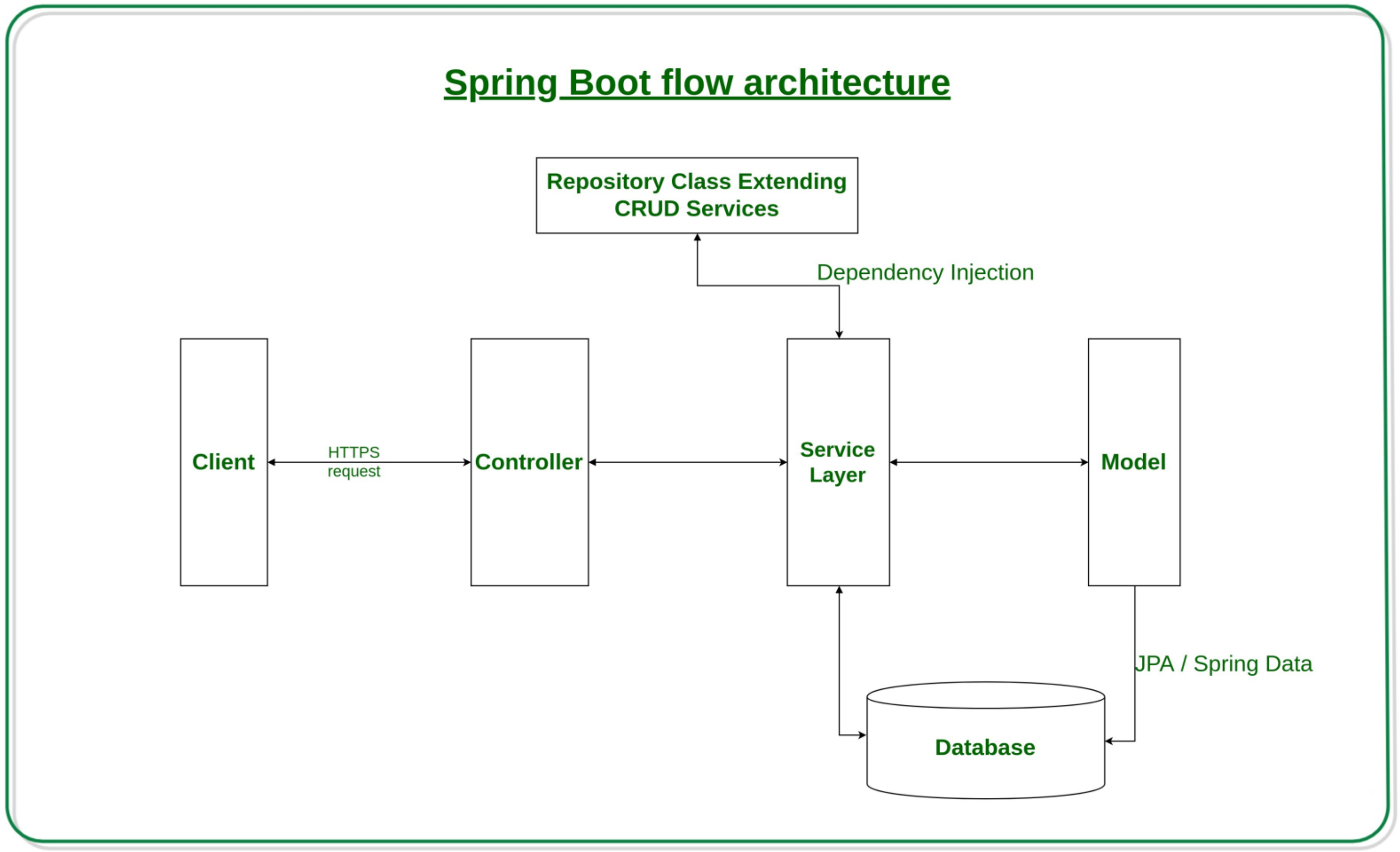
•Creates a data access layer and performs CRUD operation.

•The client makes the HTTP requests (PUT or GET).

•The request goes to the controller, and the controller maps that request and handles it. After that, it calls the service logic if required.

•In the service layer, all the business logic performs. It performs the logic on the data that is mapped to JPA with model classes.

•A JSP page is returned to the user if no error occurred.


Spring Boot flow architecture
Repository Class Extending CRUD Services
Dependency Injection
Client
HTTPS request
Controller
Service Layer
Model
JPA / Spring Data
Database

Spring Boot Annotations

- The spring boot annotations are mostly placed in org.springframework.boot.autoconfigure and org.springframework.boot.autoconfigure.condition packages. Let's learn about some frequently used spring boot annotations as well as which work behind the scene.

- @SpringBootApplication
- Spring boot is mostly about auto-configuration. This auto-configuration is done by component scanning i.e. finding all classes in classspath for @Component annotation. It also involve scanning of @Configuration annotation and initialize some extra beans.

- @SpringBootApplication annotation enable all able things in one step. It enables the three features:

- @EnableAutoConfiguration : enable auto-configuration mechanism
- @ComponentScan : enable @Component scan
- @SpringBootConfiguration : register extra beans in the context

Run the launch application and check logs

```
@SpringBootApplication
public class App
{
    public static void main(String[] args)
    {
        ApplicationContext ctx = SpringApplication.run(App.class, args);
        String[] beanNames = ctx.getBeanDefinitionNames();
        Arrays.sort(beanNames);
         for (String beanName : beanNames) {
            System.out.println(beanName);
        }
    }
}
```

CommandLineRunner

Why use CommandLineRunner interface

- Command line runners are a useful functionality to execute the various types of code that only have to be run once, right after application startup.
- FYI, Spring Batch relies on these runners in order to trigger the execution of the jobs.
- We can use the dependency injection to our advantage in order to wire in whatever dependencies that we need and in whatever way we want – in run() method implementation.

Spring boot – CommandLineRunner interface example

- How to use CommandLineRunner

- You can use CommandLineRunner interface in three ways:

- 1) Using CommandLineRunner as @Component
- 2) Implement CommandLineRunner in @SpringBootApplication
- 3) Using CommandLineRunner as Bean

1] Using CommandLineRunner as @Component

- @Component
- public class ApplicationStartupRunner implements CommandLineRunner {
-     protected final Log logger = LogFactory.getLog(getClass());
-
-     @Override
-     public void run(String... args) throws Exception {
-         logger.info("ApplicationStartupRunner run method Started !!");
-     }
- }

2) Implement CommandLineRunner in @SpringBootApplication

```
• @SpringBootApplication
• public class SpringBootWebApplication extends SpringBootServletInitializer implements CommandLineRunner {
•
•     @Override
•     protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
•         return application.sources(SpringBootWebApplication.class);
•     }
•      public static void main(String[] args) throws Exception {
•         SpringApplication.run(SpringBootWebApplication.class, args);
•     }
•      @Override
•     public void run(String... args) throws Exception {
•         logger.info("Application Started !!");
•     }
• }
```

3) Using CommandLineRunner as Bean

- You can define a bean in SpringBootApplication which return the class that implements CommandLineRunner interface.
- ApplicationStartupRunner.java

- public class ApplicationStartupRunner implements CommandLineRunner {
-     protected final Log logger = LogFactory.getLog(getClass());
-     @Override
-     public void run(String... args) throws Exception {
-         logger.info("Application Started !!");
-     }
- }

Register ApplicationStartupRunner bean

```
@SpringBootApplication
public class SpringBootWebApplication extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(SpringBootWebApplication.class);
    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(SpringBootWebApplication.class, args);
    }
    @Bean
    public ApplicationStartupRunner schedulerRunner() {
        return new ApplicationStartupRunner();
    }
}
```

- Using @Order if multiple CommandLineRunner interface implementations

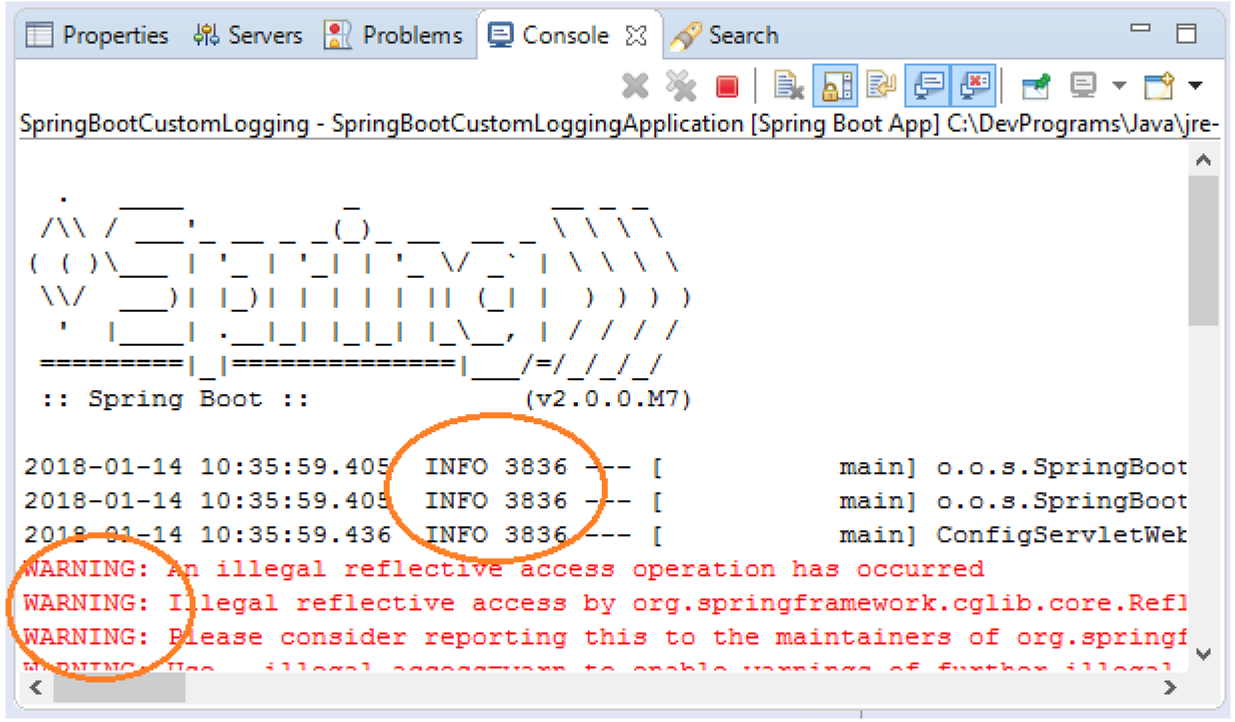
- You may have multiple implementations of CommandLineRunner interface. By default, spring boot to scan all its run() methods and execute it. But if you want to force some ordering in them, use @Order annotation.

```java
@Order(value=3)
@Component
class ApplicationStartupRunnerOne implements CommandLineRunner {
   protected final Log logger = LogFactory.getLog(getClass());
   @Override
   public void run(String... args) throws Exception {
      logger.info("ApplicationStartupRunnerOne run method Started !!");
   }
}
@Order(value=2)
@Component
class ApplicationStartupRunnerTwo implements CommandLineRunner {
   protected final Log logger = LogFactory.getLog(getClass());
   @Override
   public void run(String... args) throws Exception {
      logger.info("ApplicationStartupRunnerTwo run method Started !!");
   }
}
```

```
2017-03-08 13:55:04 - ApplicationStartupRunnerTwo run method Started !!
2017-03-08 13:55:04 - ApplicationStartupRunnerOne run method Started !!
```

Logging

- "Logging" is simply understood to be "recording" the problems during operation of applications. The problems herein are errors, warnings and other information,.... This information can be displayed on Console screen or recorded in files.
- When you run the Spring Boot application directly on Eclipse, you can see information on Console window. This information show you the state of the application, errors occurring during running application. That is Logging!.



SpringBootCustomLogging - SpringBootCustomLoggingApplication [Spring Boot App] C:\DevPrograms\Java\jre-

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.0.M7)

2018-01-14 10:35:59.405  INFO 3836 --- [           main] o.o.s.SpringBoot
2018-01-14 10:35:59.405  INFO 3836 --- [           main] o.o.s.SpringBoot
2018-01-14 10:35:59.436  INFO 3836 --- [           main] ConfigServletWeb
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.springframework.cglib.core.Refl
WARNING: Please consider reporting this to the maintainers of org.springf
WARNING: Use --illegal-access=warn to enable warnings of further illegal
```

Using the Default Spring Boot Logger

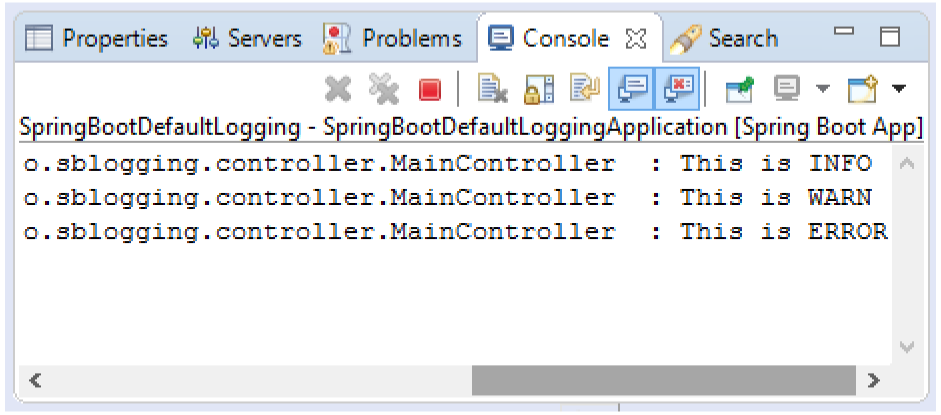
- Spring Boot comes with a preconfigured default logger based on the Logback framework. We can use this logging setup out of the box without any additional configuration since the **spring-web-starter package** we built our project on includes this dependency already.
- The default logger is great for quick prototyping or experimenting. However, we’ll inevitably want a bit more configuration, which we’ll get to in later sections.
- First, to update our logging controller to use this built-in logger, update **LoggingController.java** to the following:

- import org.slf4j.Logger;
- import org.slf4j.LoggerFactory;
- import org.springframework.web.bind.annotation.RequestMapping;
- import org.springframework.web.bind.annotation.RestController;

- @RestController
- public class LoggingController {

- Logger logger = LoggerFactory.getLogger(LoggingController.class);

- @RequestMapping("/")
- public String index() {
- logger.trace("This is a TRACE message.");
- logger.debug("This is a DEBUG message.");
- logger.info("This is an INFO message.");
- logger.warn("This is a WARN message.");
- logger.error("You guessed it, an ERROR message.");
-
- return "Welcome to Spring Logging! Check the console to see the log messages.";
- }
- }



SpringBootDefaultLogging - SpringBootDefaultLoggingApplication [Spring Boot App]

```
o.sblogging.controller.MainController  : This is INFO
o.sblogging.controller.MainController  : This is WARN
o.sblogging.controller.MainController  : This is ERROR
```

Logging Level

- Based on the severity of problem, Logback divides the information to be recorded into 5 Levels, the least severity is TRACE, and the most severity is ERROR. Note: Some Logging libraries divide the information to be recorded into 7 different levels.
- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FATAL
- OFF

By default, Spring Boot record only information with the severity of INFO or higher

```
# Default:
logging.level.root=INFO
```

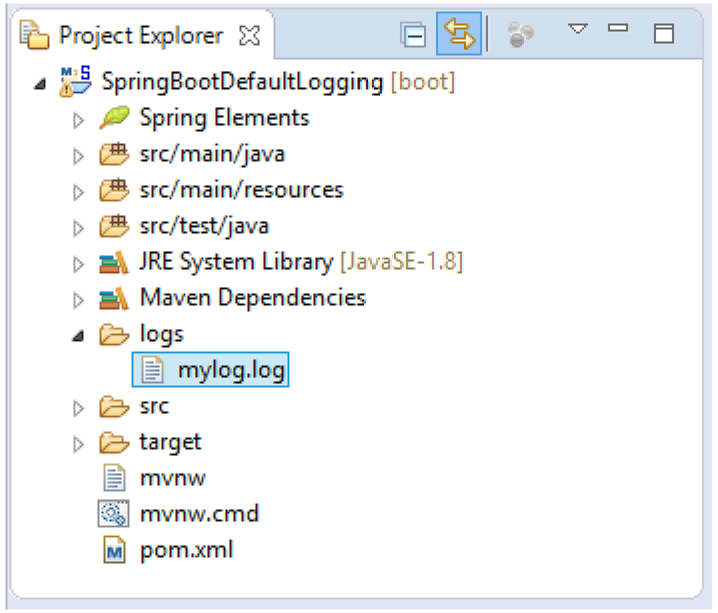
Change the Logging Level in application.properties:

```
* application.properties *
logging.level.root=WARN

# ..
```

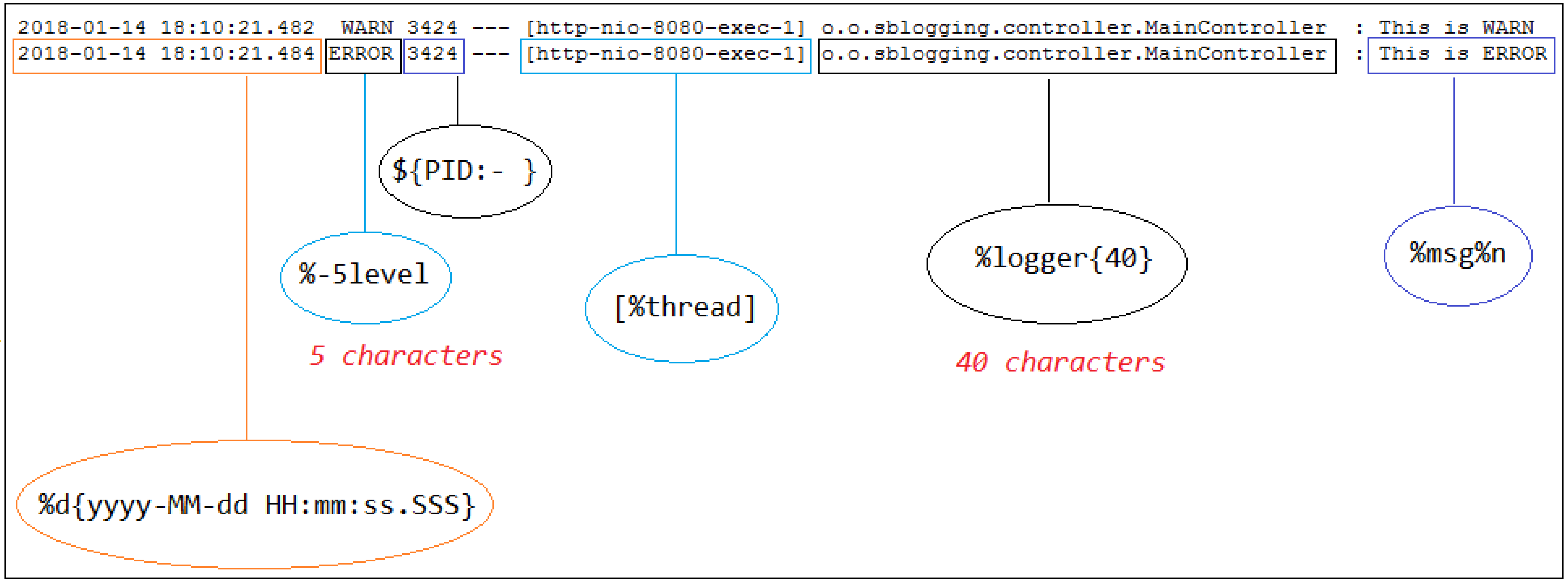
Logging File

- Default logging information is written on Console screen, however, you can configure so that they are written in files.
- * application.properties *
- ?
- logging.file=logs/mylog.log



Logging Pattern

- Log records are written in a pattern. Below is a default pattern:



```
2018-01-14 18:10:21.482  WARN 3424 --- [http-nio-8080-exec-1] o.o.sblogging.controller.MainController  : This is WARN
2018-01-14 18:10:21.484 ERROR 3424 --- [http-nio-8080-exec-1] o.o.sblogging.controller.MainController  : This is ERROR
```

- And you can change "Logging pattern" by customizing the following properties:
- logging.pattern.console
- logging.pattern.file

- # Pattern:
-
- logging.pattern.console= %d{yyyy-MMM-dd HH:mm:ss.SSS} %-5level [%thread] %logger{15} - %msg%n
-
- # Output:
-
- 2018-Jan-17 01:58:49.958 WARN [http-nio-8080-exec-1] o.o.s.c.MainController - This is WARN
- 2018-Jan-17 01:58:49.960 ERROR [http-nio-8080-exec-1] o.o.s.c.MainController - This is ERROR

- # Pattern:
-
- logging.pattern.console= %d{dd/MM/yyyy HH:mm:ss.SSS} %-5level [%thread] %logger{115} - %msg%n
-
- # Output:
-
- 17/01/2018 02:15:15.052 WARN [http-nio-8080-exec-1] org.o7planning.sblogging.controller.MainController - This is WARN
- 17/01/2018 02:15:15.054 ERROR [http-nio-8080-exec-1] org.o7planning.sblogging.controller.MainController - This is ERROR

- # Pattern:
-
- logging.pattern.console=%d{yy-MMMM-dd HH:mm:ss:SSS} %5p %t %c{2}:%L - %m%n
-
- # Output:
-
- 18-January-17 02:21:20:317 WARN http-nio-8080-exec-1 o.o.s.c.MainController:22 - This is WARN
- 18-January-17 02:21:20:320 ERROR http-nio-8080-exec-1 o.o.s.c.MainController:23 - This is ERROR

Logback Configuration Logging

- Let's see **how to include a Logback configuration** with a different color and logging pattern, with separate specifications for *console* and *file* output, and with a decent *rolling policy* to avoid generating huge log files.
- First of all, we should go toward a solution which allows handling our logging settings alone, instead of polluting *application.properties,* which is commonly used for many other application settings.
- **When a file in the classpath has one of the following names, Spring Boot will automatically load it** over the default configuration:
- *logback-spring.xml*
- *logback.xml*
- *logback-spring.groovy*
- *logback.groovy*

Create a Custom Logback Configuration

Go ahead and create the **logback-spring.xml** file under **src/main/resources** in the same place as **application.properties** and drop the following into it:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="Console" class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>
        %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
      </Pattern>
    </layout>
  </appender>

  <root level="info">
    <appender-ref ref="Console" />
  </root>

  <!-- Log everything at the TRACE level -->
  <logger name="com.example" level="trace" additivity="false">
    <appender-ref ref="Console" />
  </logger>
</configuration>
```

This message pattern uses the following Logback variables:

- **%d{yyyy-MM-dd HH:mm:ss}**—Date in the specified format
- **[%thread]** —Current thread identifier writing the message
- **%-5level** —The message level with five-character, fixed-width spacing
- **%logger{36}** —The name of the logger writing the message
- **%msg%n** —The actual message followed by a new line

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
        <Appenders>
                <Console name="Console" target="SYSTEM_OUT">
                        <PatternLayout pattern="[log4j] %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n"/>
                </Console>
        </Appenders>
        <Loggers>
                <!-- LOG everything at INFO level -->
                <Root level="info">
                        <AppenderRef ref="Console" />
                </Root>
                <!-- LOG "com.example" at TRACE level -->
                <Logger name="com.example" level="trace">
                </Logger>
        </Loggers>
</Configuration>
```

Building Web Applications

Create a Web Controller

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class GreetingController {

        @GetMapping("/greeting")
        public String greeting(@RequestParam(name="name", required=false, defaultValue="World") String name, Model model) {
                model.addAttribute("name", name);
                return "greeting";
        }

}
```

Greeting.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Getting Started: Serving Web Content</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <p th:text="'Hello, ' + ${name} + '!'" />
</body>
</html>
```

External html file

- @RequestMapping(method = RequestMethod.***GET, value = "/test")***
- **public ModelAndView welcome() {**
-     ModelAndView modelAndView = **new ModelAndView();**
-     modelAndView.setViewName("test.html");
-     **return modelAndView;**
- }

City.java

- public class City {
- private int id;
- private String name;
- private String population;
- public City(int id, String name, String population) {
- super();
- this.id = id;
- this.name = name;
- this.population = population;
- }

```
@RequestMapping(method = RequestMethod.GET, value = "/cities")
  public ModelAndView showCities(Model model) {
      ArrayList<City> params=new ArrayList<City>();
      params.add(new City(1,"Pune","Pune"));
      params.add(new City(1,"Pune","Pune"));
      params.add(new City(1,"Pune","Pune"));
      model.addAttribute("cities", params);
      return new ModelAndView("test");
  }
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Cities</title>
  <meta charset="UTF-8"/>
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
</head>
<body>
<h2>List of cities</h2>
<table>
  <tr>
    <th>Id</th>
    <th>Name</th>
    <th>Population</th>
  </tr>
  <tr th:each="city : ${cities}">
    <td th:text="${city.id}">id</td>
    <td th:text="${city.name}">name</td>
    <td th:text="${city.population}">price</td>
  </tr>
</table>
</body>
</html>
```

THYMELEAF

What is Thymeleaf?

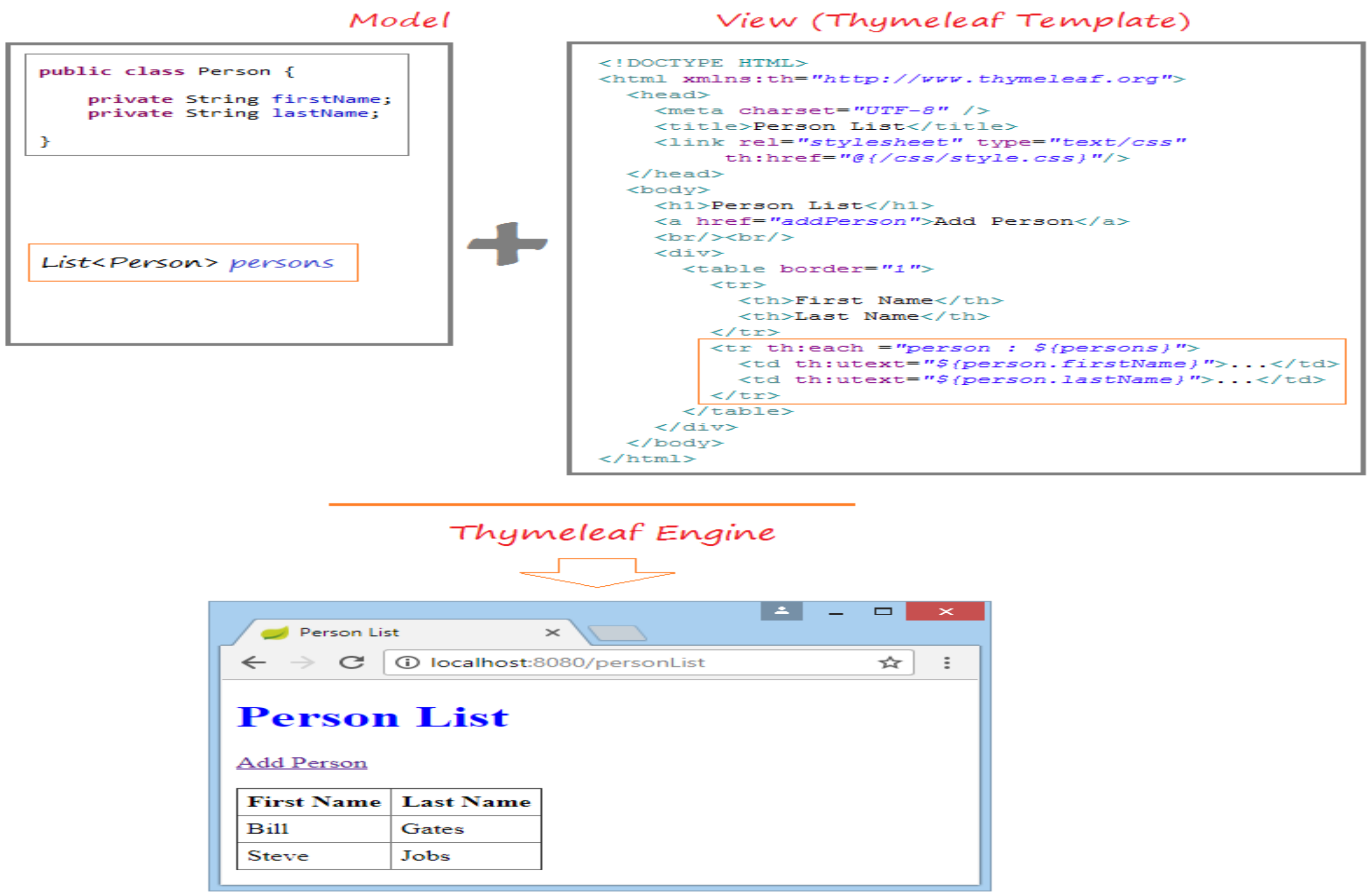
- The **Thymeleaf** is an open-source Java library that is licensed under the **Apache License 2.0**. It is a **HTML5/XHTML/XML** template engine. It is a **server-side Java template** engine for both web (servlet-based) and non-web (offline) environments. It is perfect for modern-day HTML5 JVM web development. It provides full integration with Spring Framework.
- It applies a set of transformations to template files in order to display data or text produced by the application. It is appropriate for serving XHTML/HTML5 in web applications.
- The goal of Thymeleaf is to provide a **stylish** and **well-formed** way of creating templates. It is based on XML tags and attributes. These XML tags define the execution of predefined logic on the DOM (Document Object Model) instead of explicitly writing that logic as code inside the template. It is a substitute for **JSP**.
- The architecture of Thymeleaf allows the **fast processing** of templates that depends on the caching of parsed files. It uses the least possible amount of I/O operations during execution

What kind of templates can the Thymeleaf process?

- Thymeleaf can process six types of templates (also known as **Template Mode**) are as follows:
- XML
- Valid XML
- XHTML
- Valid XHTML
- HTML5
- Legacy HTML5

Thymeleaf Features

- It works on both web and non-web environments.
- Java template engine for HTML5/ XML/ XHTML.
- Its high-performance parsed template cache reduces I/O to the minimum.
- It can be used as a template engine framework if required.
- It supports several template modes: XML, XHTML, and HTML5.
- It allows developers to extend and create custom dialect.
- It is based on modular features sets called dialects.
- It supports internationalization.



```
public class Person {

    private String firstName;
    private String lastName;
}
```

```
List<Person> persons
```

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>Person List</title>
    <link rel="stylesheet" type="text/css"
         th:href="@{/css/style.css}"/>
  </head>
  <body>
    <h1>Person List</h1>
    <a href="addPerson">Add Person</a>
    <br/><br/>
    <div>
      <table border="1">
        <tr>
          <th>First Name</th>
          <th>Last Name</th>
        </tr>
        <tr th:each ="person : ${persons}">
          <td th:utext="${person.firstName}">...</td>
          <td th:utext="${person.lastName}">...</td>
        </tr>
      </table>
    </div>
  </body>
</html>
```

Person List

localhost:8080/personList

Person List

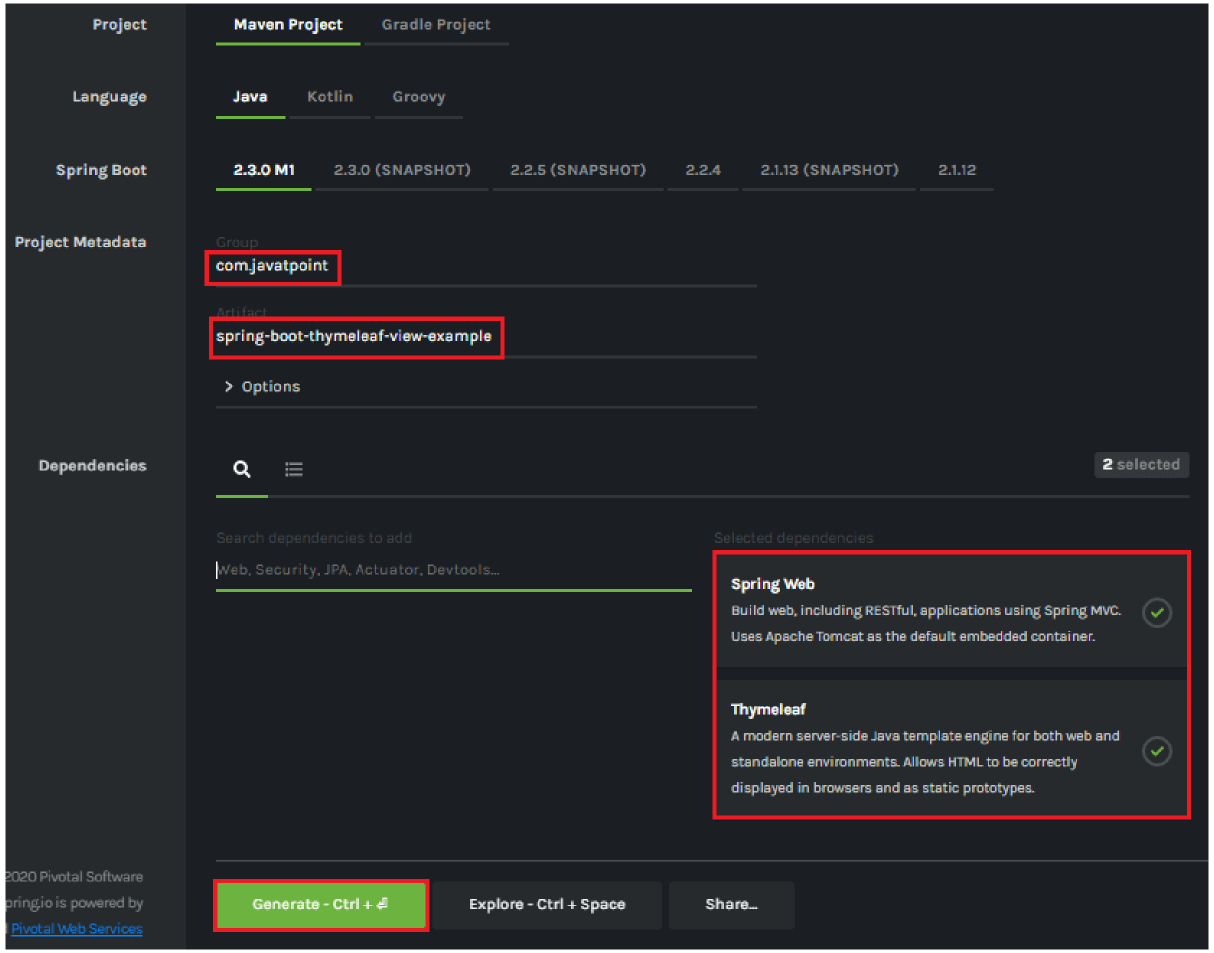
Add Person

First Name	Last Name
Bill	Gates
Steve	Jobs

- **Thymeleaf** can be used to replace **JSP** on the **View** Layer of **Web MVC** application. **Thymeleaf** is open source code software, licensed under the **Apache 2.0**.

Example 0

Example 1



Project: [✓ Maven Project] Gradle Project

Language: [✓ Java] Kotlin Groovy

Spring Boot: [✓ 2.3.0 M1] 2.3.0 (SNAPSHOT) 2.2.5 (SNAPSHOT) 2.2.4 2.1.13 (SNAPSHOT) 2.1.12

Project Metadata

Group: com.javatpoint

Artifact: spring-boot-thymeleaf-view-example

> Options

Dependencies — 2 selected

Search dependencies to add: Web, Security, JPA, Actuator, Devtools...

Selected dependencies

Spring Web
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container. ✓

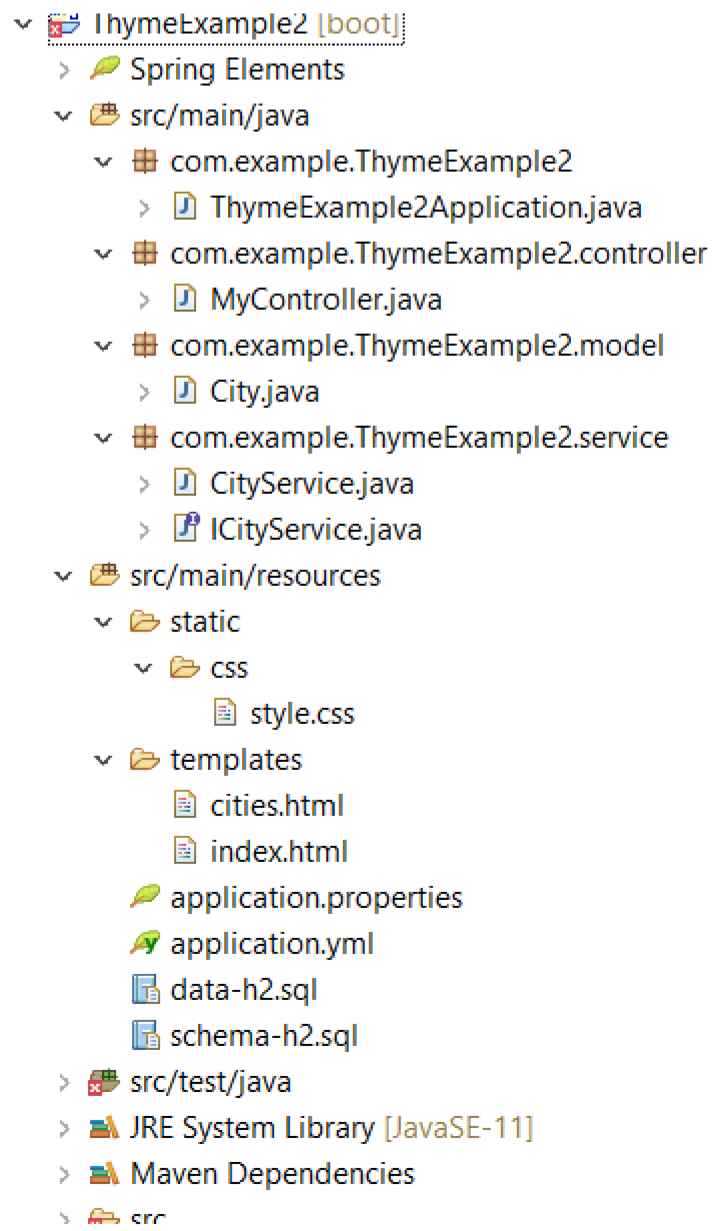
Thymeleaf
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes. ✓

2020 Pivotal Software
pring.io is powered by
Pivotal Web Services

- C:\Users\Kranti\Desktop\WEB and Spring Boot\DAY 4\Thymeleaf Example 1.docx

- Eclipse
- C:\Users\Kranti\Downloads\spring-boot-thymeleaf-view-example

Example 2



```
ThymeExample2 [boot]
  Spring Elements
  src/main/java
    com.example.ThymeExample2
      ThymeExample2Application.java
    com.example.ThymeExample2.controller
      MyController.java
    com.example.ThymeExample2.model
      City.java
    com.example.ThymeExample2.service
      CityService.java
      ICityService.java
  src/main/resources
    static
      css
        style.css
    templates
      cities.html
      index.html
    application.properties
    application.yml
    data-h2.sql
    schema-h2.sql
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
  src
```

Spring Boot - Rest Template

What is REST?

- REST stands for REpresentational State Transfer. REST specifies a set of architectural constraints. Any service which satisfies these constraints is called RESTful Service.
- The five important constraints for RESTful Web Service are
- Client - Server : There should be a service producer and a service consumer.
- The interface (URL) is uniform and exposing resources.
- The service is stateless.
- The service results should be Cacheable. HTTP cache, for example.
- Service should assume a Layered architecture. Client should not assume direct connection to server - it might be getting info from a middle layer - cache.

- Spring Boot RestTemplate tutorial shows how to use RestTemplate to create synchronous HTTP requests in a Spring application.
- RestTemplate is a synchronous client to perform HTTP requests. It uses a simple, template method API over underlying HTTP client libraries such as the JDK HttpURLConnection, Apache HttpComponents, and others.
- Since Spring 1.5.0, a new client WebClient is available that can be use do create both synchronous and asynchronous requests. In the future releases, RestTemplate will be deprecated in favour of WebClient.
- Rest Template is used to create applications that consume RESTful Web Services. You can use the **exchange()** method to consume the web services for all HTTP methods. The code given below shows how to create Bean for Rest Template to auto wiring the Rest Template object.
-

Spring Boot RestTemplate example

- In the following application we create a custom test server that produces JSON data and use RestTemplate to generate a HTTP request and consume the returned JSON data.

- Creating JSON server
- We use Node to create a JSON test server for our purposes.

- $ node --version
- v11.2.0
- We show the version of Node.

- $ npm init
- $ npm i -g json-server
- $ npm i faker fs

```
const faker = require('faker')
const fs = require('fs')
function generateUsers() {
  let users = []
  for (let id=1; id <= 100; id++) {
    let firstName = faker.name.firstName()
    let lastName = faker.name.lastName()
    let email = faker.internet.email()
    users.push({
      "id": id,
      "first_name": firstName,
      "last_name": lastName,
      "email": email
    })
  }
  return { "users": users }
}
let dataObj = generateUsers();
fs.writeFileSync('data.json', JSON.stringify(dataObj, null, '\t'));
```

With faker we generate one hundred users with id, first name, last name, and email attributes. The data is written to data.json file. The file is used by json-server.

```
$ node generate_fake_users.js
```

We generate one hundred fake users.

```
$ json-server --watch data.json

\{^_^}/ hi!

Loading data.json
Done

Resources
http://localhost:3000/users

Home
http://localhost:3000
```

We start the json-server. Now we can create a request to the http://localhost:3000/users resource to get one hundred users in JSON.

- Spring Boot application
- We create a Spring Boot application. We need the following Maven dependencies and plugins: spring-boot-starter, spring-web, jackson-databind, spring-boot-starter-test, and spring-boot-maven-plugin.

- application.properties
- spring.main.banner-mode=off
- logging.level.root=INFO
- logging.pattern.console=%d{dd-MM-yyyy HH:mm:ss} %magenta([%thread]) %highlight(%-5level) %logger.%M - %msg%n

- myrest.url=http://localhost:3000/users

```java
import com.fasterxml.jackson.annotation.JsonProperty;
public class User {
  private int id;
  private String firstName;
  private String lastName;
  private String email;
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
 public String getFirstName() {
    return firstName;
  }
  @JsonProperty("first_name")
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
```

```java
  public String getLastName() {
    return lastName;
  }

  @JsonProperty("last_name")
  public void setLastName(String lastName) {

    this.lastName = lastName;
  }

  public String getEmail() {
    return email;
  }

  public void setEmail(String email) {

    this.email = email;
  }

  @Override
  public String toString() {

    final var sb = new StringBuilder("User{");
    sb.append("id=").append(id);
    sb.append(", firstName='").append(firstName).append('\'');
    sb.append(", lastName='").append(lastName).append('\'');
    sb.append(", email='").append(email).append('\'');
    sb.append('}');

    return sb.toString();
  }
}
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class MyRestService {
   @Autowired
   private RestTemplate myRestTemplate;
   @Value("${myrest.url}")
   private String restUrl;
   public User[] getUsers() {
      var users = myRestTemplate.getForObject(restUrl, User[].class);
       return users;
   }
}
```

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);
    }
}
```

Spring Boot JPA

- **Spring Boot JPA** is a Java specification for managing **relational** data in Java applications. It allows us to access and persist data between Java object/ class and relational database. JPA follows **Object-Relation Mapping** (ORM). It is a set of interfaces. It also provides a runtime **EntityManager** API for processing queries and transactions on the objects against the database. It uses a platform-independent object-oriented query language JPQL (Java Persistent Query Language).
- In the context of persistence, it covers three areas:
- The Java Persistence API
- **Object-Relational** metadata
- The API itself, defined in the **persistence** package
- JPA is not a framework. It defines a concept that can be implemented by any framework.

Why should we use JPA?

- JPA is simpler, cleaner, and less labor-intensive than JDBC, SQL, and hand-written mapping. JPA is suitable for non-performance oriented complex applications. The main advantage of JPA over JDBC is that, in JPA, data is represented by objects and classes while in JDBC data is represented by tables and records. It uses POJO to represent persistent data that simplifies database programming. There are some other advantages of JPA:
- JPA avoids writing DDL in a database-specific dialect of SQL. Instead of this, it allows mapping in XML or using Java annotations.
- JPA allows us to avoid writing DML in the database-specific dialect of SQL.
- JPA allows us to save and load Java objects and graphs without any DML language at all.
- When we need to perform queries JPQL, it allows us to express the queries in terms of Java entities rather than the (native) SQL table and columns.

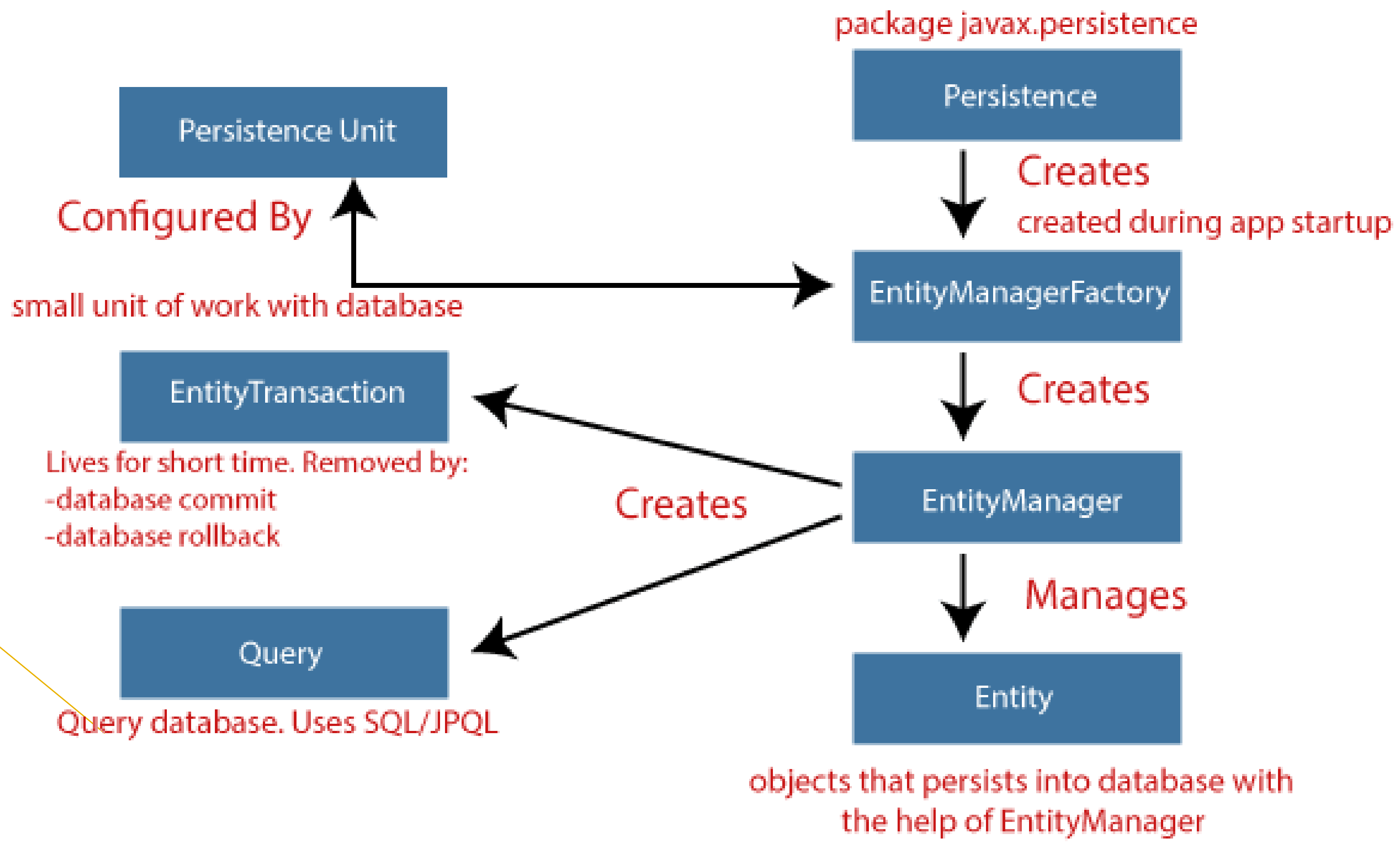
JPA Features

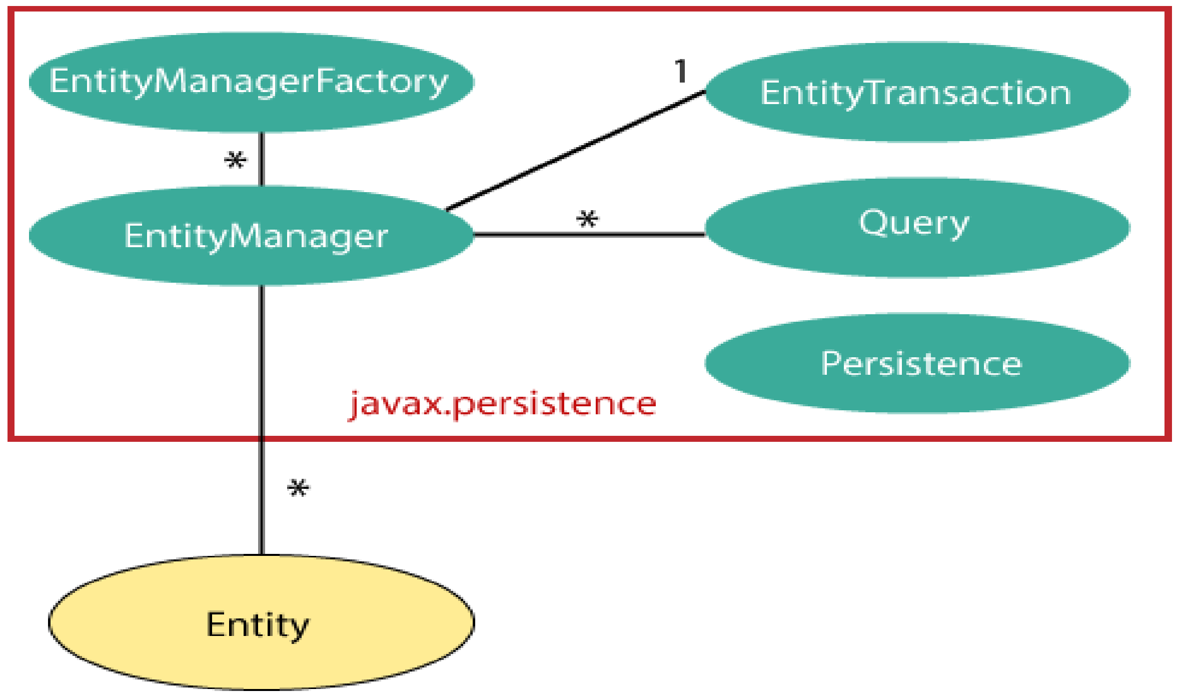
- There are following features of JPA:
- It is a powerful repository and custom **object-mapping abstraction.**
- It supports for **cross-store persistence**. It means an entity can be partially stored in MySQL and Neo4j (Graph Database Management System).
- It dynamically generates queries from queries methods name.
- The domain base classes provide basic properties.
- It supports transparent auditing.
- Possibility to integrate custom repository code.
- It is easy to integrate with Spring Framework with the custom namespace.

JPA Architecture

- JPA is a source to store business entities as relational entities. It shows how to define a POJO as an entity and how to manage entities with relation.
- The following figure describes the class-level architecture of JPA that describes the core classes and interfaces of JPA that is defined in the **javax persistence** package. The JPA architecture contains the following units:
- **Persistence:** It is a class that contains static methods to obtain an EntityManagerFactory instance.
- **EntityManagerFactory:** It is a factory class of EntityManager. It creates and manages multiple instances of EntityManager.
- **EntityManager:** It is an interface. It controls the persistence operations on objects. It works for the Query instance.
- **Entity:** The entities are the persistence objects stores as a record in the database.
- **Persistence Unit:** It defines a set of all entity classes. In an application, EntityManager instances manage it. The set of entity classes represents the data contained within a single data store.
- **EntityTransaction:** It has a **one-to-one** relationship with the EntityManager class. For each EntityManager, operations are maintained by EntityTransaction class.
- **Query:** It is an interface that is implemented by each JPA vendor to obtain relation objects that meet the criteria.

Architecture of Java Persistence API


package javax.persistence
Persistence
Creates
created during app startup
EntityManagerFactory
Creates
EntityManager
Manages
Entity
objects that persists into database with the help of EntityManager
Persistence Unit
Configured By
small unit of work with database
EntityTransaction
Lives for short time. Removed by:
-database commit
-database rollback
Creates
Query
Query database. Uses SQL/JPQL

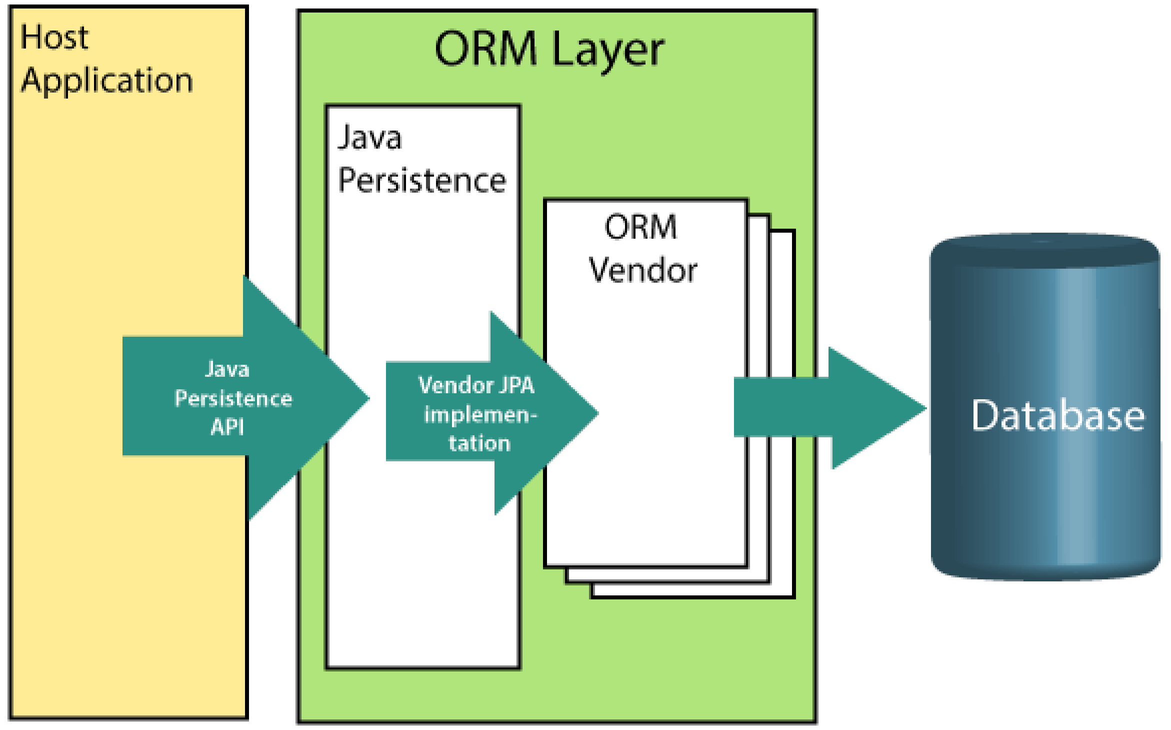


JPA Class Relationship

•The relationship between EntityManager and EntiyTransaction is **one-to-one**. There is an EntityTransaction instance for each EntityManager operation.
•The relationship between EntityManageFactory and EntiyManager is **one-to-many**. It is a factory class to EntityManager instance.
•The relationship between EntityManager and Query is **one-to-many**. We can execute any number of queries by using an instance of EntityManager class.
•The relationship between EntityManager and Entity is **one-to-many**. An EntityManager instance can manage multiple Entities.

Object-Relation Mapping (ORM)

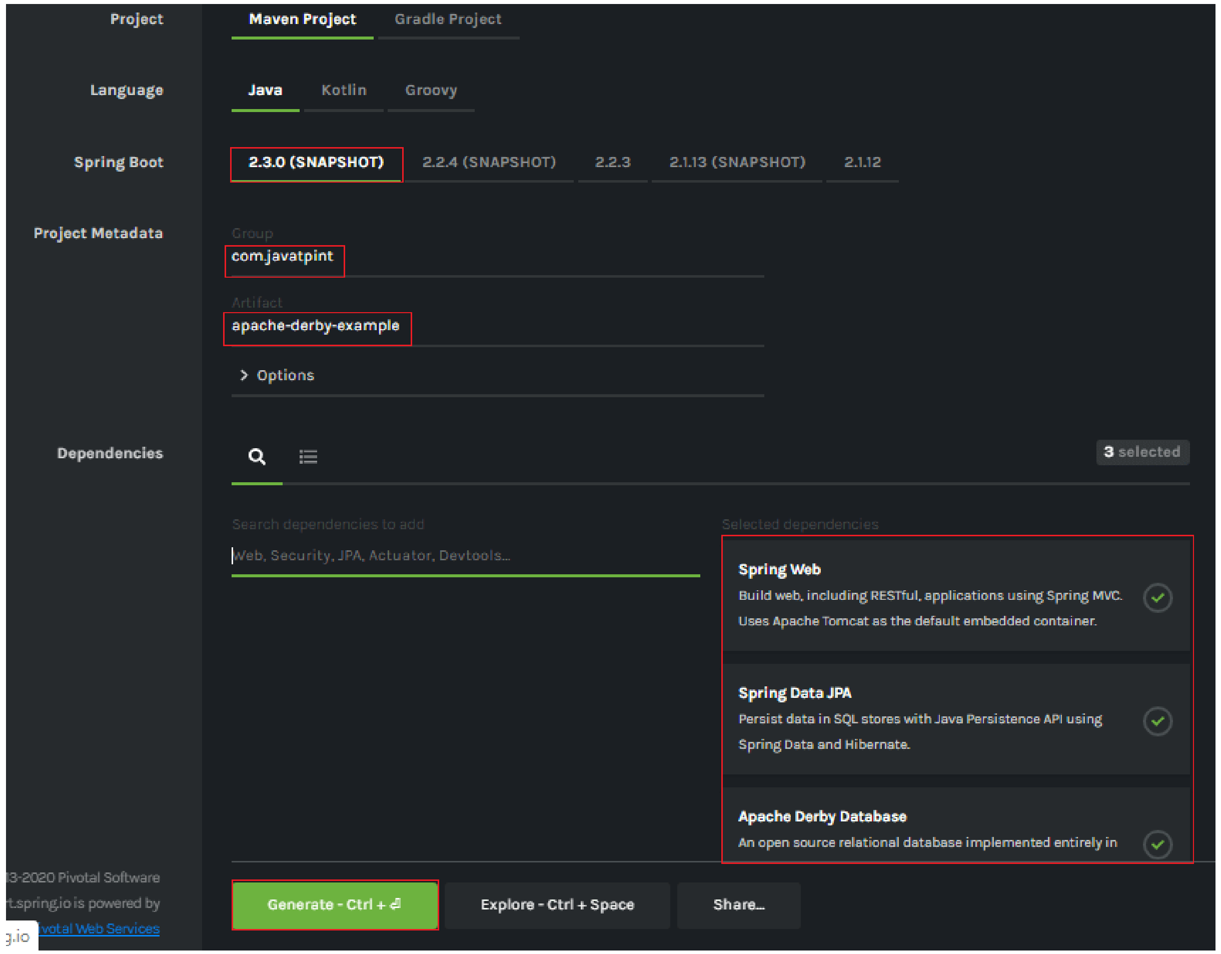
- In ORM, the mapping of Java objects to database tables, and vice-versa is called **Object-Relational Mapping.** The ORM mapping works as a bridge between a **relational database** (tables and records) and **Java application** (classes and objects).
- In the following figure, the ORM layer is an adapter layer. It adapts the language of object graphs to the language of SQL and relation tables



JPA	Hibernate
JPA is a **Java specification** for mapping relation data in Java application.	Hibernate is an **ORM framework** that deals with data persistence.
JPA does not provide any implementation classes.	It provides implementation classes.
It uses platform-independent query language called **JPQL** (Java Persistence Query Language).	It uses its own query language called **HQL** (Hibernate Query Language).
It is defined in **javax.persistence** package.	It is defined in **org.hibernate** package.
It is implemented in various ORM tools like **Hibernate, EclipseLink,** etc.	Hibernate is the **provider** of JPA.
JPA uses **EntityManager** for handling the persistence of data.	In Hibernate uses **Session** for handling the persistence of data.

Spring Boot Starter Data JPA

- Spring Boot provides starter dependency **spring-boot-starter-data-jpa** to connect Spring Boot application with relational database efficiently. The spring-boot-starter-data-jpa internally uses the spring-boot-jpa dependency.
- **<dependency>**
- **<groupId>**org.springframework.boot**</groupId>**
- **<artifactId>**spring-boot-starter-data-jpa**</artifactId>**
- **<version>**2.2.2.RELEASE**</version>**
- **</dependency>**



Spring Web
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Apache Derby Database
An open source relational database implemented entirely in

3-2020 Pivotal Software
t.spring.io is powered by
ivotal Web Services
g.io

- **Step 1:** Open Spring Initializr https://start.spring.io/.
- **Step 2:** Select the latest version of Spring Boot **2.3.0(SNAPSHOT)**
- **Step 3:** Provide the **Group** name. We have provided **com.javatpoint.**
- **Step 4:** Provide the **Artifact** Id. We have provided **apache-derby-example**.
- **Step 5:** Add the dependencies: **Spring Web, Spring Data JPA,** and **Apache Derby Database**.
- **Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps the project in a Jar file and downloads it to the local system.

Step 7: Extract the Jar file and paste it into the STS workspace.

- **Step 8: Import** the project folder into STS.
- File -> Import -> Existing Maven Projects -> Browse -> Select the folder apache-derby-example -> Finish
- It takes some time to import.
- **Step 9:** Create a package with the name **com.javatpoint.model** in the folder **src/main/java.**
- **Step 10:** Create a class with the name **UserRecord** in the package **com.javatpoint.model** and do the following:
- Define three variables **id, name,** and **email**.
- Generate Getters and Setter.
  Right-click on the file -> Source -> Generate Getters and Setters
- Define a default constructor.
- Mark the class as an **Entity** by using the annotation **@Entity.**
- Mark **Id** as the primary key by using the annotation **@Id.**

Spring Boot JDBC Example

- // application.properties
- spring.datasource.url=jdbc:mysql://localhost:3306/springbootdb
- spring.datasource.username=root
- spring.datasource.password=mysql
- spring.jpa.hibernate.ddl-auto=create-drop

SpringBootJdbcApplication

```
• import org.springframework.boot.SpringApplication;
• import org.springframework.boot.autoconfigure.SpringBootApplication;
• @SpringBootApplication
• public class SpringBootJdbcApplication {
•     public static void main(String[] args) {
•         SpringApplication.run(SpringBootJdbcApplication.class, args);
•     }
• }
```

SpringBootJdbcController

- **import** org.springframework.web.bind.annotation.RequestMapping;
- **import** org.springframework.beans.factory.annotation.Autowired;
- **import** org.springframework.jdbc.core.JdbcTemplate;
- **import** org.springframework.web.bind.annotation.RestController;
- @RestController
- **public class** SpringBootJdbcController {
-     @Autowired
-     JdbcTemplate jdbc;
-     @RequestMapping("/insert")
-     **public** String index(){
-         jdbc.execute("insert into user(name,email)values('java','java@gmail.com')");
-         **return**"data inserted Successfully";
-     }
- }

MongoDB Database

- spring.data.mongodb.database=gift_shop
- spring.data.mongodb.host=localhost
- spring.data.mongodb.port=27017

Customer.java

- @Id
- **public int id;**
- **public String firstname;**
- **public String lastname;**

- package com.example.demo;
- import org.springframework.data.mongodb.repository.MongoRepository;
- public interface CustomerRepository extends MongoRepository<Customer,String>{
- }

```java
package com.example.demo;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MongodbappApplication implements CommandLineRunner{

        @Autowired

        private CustomerRepository customerRepository;

        public static void main(String[] args) {

                SpringApplication.run(MongodbappApplication.class, args);

        }

        @Override

        public void run(String... args) throws Exception {

                Customer c1=new Customer(1,"Esan","Pune");

                Customer c2=new Customer(2,"Esan","Pune");

                Customer c3=new Customer(3,"Esan","Pune");

                Customer c4=new Customer(4,"tani","Pune");

                customerRepository.save(c1);

                customerRepository.save(c2);

                customerRepository.save(c3);

                customerRepository.save(c4);


                System.out.println("******************");

                List<Customer> customer=customerRepository.findAll();

                for(Customer c : customer) {

                        System.out.println(c.toString());

                }



}
```

Testing in Spring Boot 2

- JUnit is the most popular Java Unit testing framework
- We typically work in large projects - some of these projects have more than 2000 source files or sometimes it might be as big as 10000 files with one million lines of code.
- Before unit testing, we depend on deploying the entire app and checking if the screens look great. But that’s not very efficient. And it is manual.
- Unit Testing focuses on writing automated tests for individual classes and methods.
- JUnit is a framework which will help you call a method and check (or assert) whether the output is as expected.
- The important thing about automation testing is that these tests can be run with continuous integration - as soon as some code changes.

- The spring-boot-starter-test “Starter” (in the test scope) contains the following provided libraries:

- JUnit: The de-facto standard for unit testing Java applications.
- Spring Test & Spring Boot Test: Utilities and integration test support for Spring Boot applications.
- AssertJ: A fluent assertion library.
- Hamcrest: A library of matcher objects (also known as constraints or predicates).
- Mockito: A Java mocking framework.
- JSONassert: An assertion library for JSON.
- JsonPath: XPath for JSON.

```
@SpringBootTest
public class SmokeTest {
	@Autowired
	private HomeController controller;
	@Test
	public void contexLoads() throws Exception {
		assertThat(controller).isNotNull();
	}
}
```

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HttpRequestTest {
	@LocalServerPort
	private int port;
	@Autowired
	private TestRestTemplate restTemplate;
	@Test
	public void greetingShouldReturnDefaultMessage() throws Exception {
		assertThat(this.restTemplate.getForObject("http://localhost:" + port +
"/",
				String.class)).contains("Hello, World");
	}
}
```

```
@SpringBootTest
@AutoConfigureMockMvc
public class TestingWebApplicationTest {
        @Autowired
        private MockMvc mockMvc;
        @Test
        public void shouldReturnDefaultMessage() throws Exception {
                this.mockMvc.perform(get("/")).andDo(print()).andExpect(status().isOk())
                                .andExpect(content().string(containsString("Hello, World")));
        }
}
```

```
@WebMvcTest
public class WebLayerTest {

        @Autowired
        private MockMvc mockMvc;
        @Test
        public void shouldReturnDefaultMessage() throws Exception {
                this.mockMvc.perform(get("/")).andDo(print()).andExpect(status().isOk())
                                .andExpect(content().string(containsString("Hello, World")));
        }
}
```

```java
@Controller
public class GreetingController {

        private final GreetingService service;

        public GreetingController(GreetingService service) {
                this.service = service;
        }

        @RequestMapping("/greeting")
        public @ResponseBody String greeting() {
                return service.greet();
        }

}
```

```
import org.springframework.stereotype.Service;

@Service
public class GreetingService {
    public String greet() {
        return "Hello, World";
    }
}
```

```
@WebMvcTest(GreetingController.class)
public class WebMockTest {
	@Autowired
	private MockMvc mockMvc;
	@MockBean
	private GreetingService service;
	@Test
	public void greetingShouldReturnMessageFromService() throws Exception {
		when(service.greet()).thenReturn("Hello, Mock");
		this.mockMvc.perform(get("/greeting")).andDo(print()).andExpect(status().isOk())
				.andExpect(content().string(containsString("Hello, Mock")));
	}
}
```