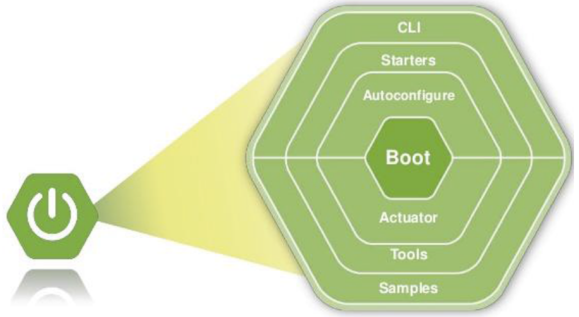
Spring Boot - The evolution…



Spring Boot definition

- Spring Boot makes it easy to create stand-alone, production-grade Spring based applications that you can "just run".
- We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration
- Spring Boot enables developers to focus on the business logic behind their microservice. It aims to take care of all the nitty-gritty technical details involved in developing microservices.

Primary Goals of Spring Boot

- Enable quickly getting off the ground with Spring-based projects.
- Be opinionated. Make default assumptions based on common usage. Provide configuration options to handle deviations from defaults.
- Provide a wide range of nonfunctional features out of the box.
- Do not use code generation and avoid using a lot of XML configuration.

Non Functional Features of Spring Boot

- Default handling of versioning and configuration of a wide range of frameworks, servers, and specifications
- Default options for application security
- Default application metrics with possibilities to extend
- Basic application monitoring using health checks
- Multiple options for externalized configuration

Spring Initializr

- Spring Initializr provides a lot of flexibility in creating projects. You have options to do the following:
  - Choose your build tool: Maven or Gradle.
  - Choose the Spring Boot version you want to use.
  - Configure a Group ID and Artifact ID for your component.
  - Choose the starters (dependencies) that you would want for your project.
  - Choose how to package your component: JAR or WAR.
  - Choose the Java version you want to use.
  - Choose the JVM language you want to use.

Hello with Spring Boot

- We will start with building our first Spring Boot application.
- We will use Maven to manage dependencies.
- The following steps are involved in starting up with a Spring Boot application:
  - Configure spring-boot-starter-parent in your pom.xml file.
  - Configure the pom.xml file with the required starter projects.
  - Configure spring-boot-maven-plugin to be able to run the application.
  - Create your first Spring Boot launch class.

spring-boot-starter-parent

- The spring-boot-starter-parent dependency is the parent POM providing dependency and plugin management for Spring Boot-based applications.
- A spring-boot-starter-parent dependency contains the default versions of Java to use, the default versions of dependencies that Spring Boot uses, and the default configuration of the Maven plugins.

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version> 2.1.1.RELEASE</version>
<relativePath/>
</parent>
```

SpringApplication

- The SpringApplication class can be used to Bootstrap and launch a Spring application from a Java main method.
- The following are the steps that are typically performed when a Spring Boot application is bootstrapped:
  - Create an instance of Spring's ApplicationContext.
  - Enable the functionality to accept command-line arguments and expose them as Spring properties.
  - Load all the Spring beans as per the configuration.

```
@SpringBootApplication
public class HelloApplication {

public static void main(String[] args) {
SpringApplication.run(HelloApplication.class, args);
}
}
```

SpringApplication

- Classes that can be used to bootstrap and launch a Spring application from a Java main method. By default class will perform the following steps to bootstrap your application:
  - Create an appropriate ApplicationContext instance
  - Register a CommandLinePropertySource to expose command line arguments as Spring properties
  - Refresh the application context, loading all singleton beans
  - Trigger any CommandLineRunner beans

```java
@SpringBootApplication
public class HelloApplication {

public static void main(String[] args) {
SpringApplication.run(HelloApplication.class, args);
}
}
```

@SpringBootApplication

- The **@SpringBootApplication** annotation is a shortcut for three annotations:
  - **@Configuration**: Indicates that this a Spring application context configuration file.
  - **@EnableAutoConfiguration**: Enables auto-configuration, an important feature of Spring Boot. We will discuss auto-configuration later in a separate section.
  - **@ComponentScan**: Enables scanning for Spring beans in the package of this class and all its sub packages.

```
@SpringBootApplication
public class HelloApplication {

public static void main(String[] args) {
SpringApplication.run(HelloApplication.class, args);
}
}
```

@SpringBootApplication Contd..

- Let’s see what all beans are loaded at this point
- Where are these beans defined? &How are these beans created?
- That's the magic of Spring auto-configuration.
- Whenever we add a new dependency to a Spring Boot project, Spring Boot auto-configuration automatically tries to configure the beans based on the dependency.

```
public static void main(String[] args) {
ConfigurableApplicationContext context =
SpringApplication.run(HelloApplication.class, args);
String[] beanNames = context.getBeanDefinitionNames();
Arrays.sort(beanNames); //optional
    for (String beanName : beanNames) {
        System.out.println(beanName);
    }
}
```

@Component

- The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context.
- To use this annotation, apply it over class as below:

```
@Component
public class MyHello {
	public MyHello() {
		System.out.println("hello!");
	}
}
```

- This should print in console, which indicates this component was instantiated, we can reprint beans and validate.

@Component vs @Bean

- @Component and @Bean do two quite different things, and shouldn't be confused.
- @Component (and @Service and @Repository) are used to **auto-detect and auto-configure** beans using classpath scanning. There's an implicit one-to-one mapping between the annotated class and the bean (i.e. one bean per class). Control of wiring is quite limited with this approach, since it's purely declarative.
- @Bean is used to ***explicitly*** declare a single bean, rather than letting Spring do it automatically as above. It decouples the declaration of the bean from the class definition, and lets you create and configure beans exactly how you choose.

@Component vs @Bean contd..

- **@Component** Preferable for component scanning and automatic wiring.
- *When should you use **@Bean**?*
  - Sometimes automatic configuration is not an option. **When?** Let's imagine that you want to wire components from 3rd-party libraries (you don't have the source code so you can't annotate its classes with @Component), so automatic configuration is not possible.
  - The **@Bean** annotation **returns an object** that spring should register as bean in application context. The **body of the method** bears the logic responsible for creating the instance.

Inject from application.properties

- Add a property and let’s inject
- message= Hello from Spring Boot in application.properties

```
@Component
public class MyHello {
// inject via application.properties / 
application.yml
@Value("${message}")
private String message;

public MyHello() {
System.out.println("hello!");
}

public String getMessage() {
return message;
}
}
```

CommandLineRunner (to show DI)

- Interface used to indicate that a bean should *run* when it is contained within a SpringApplication.

```java
@SpringBootApplication
public class HelloApplication implements
CommandLineRunner {

@Autowired
private MyHello myHello;

public static void main(String[] args) {
SpringApplication.run(HelloApplication.class, args);
System.out.println("Reaching end!");
}

@Override
public void run(String... args) throws Exception {
         System.out.println(myHello.getMessage());
}
}
```

@ Autowired (to show DI)

- The **@Autowired** annotation provides more fine-grained control over where and how autowiring should be accomplished.
- The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

Profiles

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

```
@Component
@Profile("dev")
public class MyHello {

@Value("${message}")
private String message;

public String getMessage() {
return message;
}
}
```

Application.yml

- Using YAML Instead of Properties files is better.
- YAML is a superset of JSON and, as such, is a convenient format for specifying hierarchical configuration data.
- The SpringApplication class automatically supports YAML as an alternative to properties whenever you have the SnakeYAML library on your classpath.
- You can specify multiple profile-specific YAML documents in a single file by using a **spring.profiles** key to indicate when the document applies.
- In addition to application.properties files, profile-specific properties can also be defined by using the following naming convention: **application-{profile}.properties**.
- The Environment has a set of default profiles (by default, [default]) that are used if no active profiles are set. In other words, if no profiles are explicitly activated, then properties from **application-default.properties** are loaded.

YML example

- You can specify multiple profile-specific YAML documents in a single file by using a spring.profiles key to indicate when the document applies, as shown in the following example:

```yaml
spring:
  profiles:
    active: dev
---
spring:
  profiles: dev
message: Hello from Spring Boot Dev

---
spring:
  profiles: test
message: Hello from Spring Boot Test
```

Switching Profiles

- There are many way to switch profiles, below is the command line version:
- **JVM System Parameter**
  - The profile names can also be passed in via a JVM system parameter. The profile names passed as the parameter will be activated during application start-up:
  - -Dspring.profiles.active=test
- **Environment Variable**
  - In a Unix environment, profiles can also be activated via the environment variable:
  - export spring_profiles_active=dev

Starters in Spring Boot

- Starters are simplified dependency descriptors customized for different purposes.
- For example, **spring-boot-starter-web** is the starter for building web application, including RESTful, using Spring MVC. It uses Tomcat as the default embedded container.
- If I want to develop a web application using Spring MVC, all we would need to do is include spring-boot-starter-web in our dependencies, and we get the following automatically pre-configured:
  - Spring MVC
  - Compatible versions of jackson-databind (for binding) and hibernate-validator (for form validation)
  - spring-boot-starter-tomcat (starter project for Tomcat)

What other starters are there?

Starter	Description
spring-boot-starter-webservices	This is a starter project to develop XMLbased web services.
spring-boot-starter-web	This is a starter project to build Spring MVC-based web applications or RESTful applications. It uses Tomcat as the default embedded servlet container.
spring-boot-starter-activemq	This supports message-based communication using JMS on ActiveMQ.
spring-boot-starterintegration	This supports the Spring Integration Framework that provides implementations for Enterprise Integration Patterns.
spring-boot-starter-test	This provides support for various unit testing frameworks, such as JUnit, Mockito, and Hamcrest matchers.
spring-boot-starter-jdbc	This provides support for using Spring JDBC. It configures a Tomcat JDBC connection pool by default.
spring-boot-startervalidation	This provides support for the Java Bean Validation API. Its default implementation is hibernate-validator.
spring-boot-starter-hateoas	HATEOAS stands for Hypermedia as the Engine of Application State. RESTful services that use HATEOAS return links to additional resources that are related to the current context in addition to data.
spring-boot-starter-jersey	JAX-RS is the Java EE standard to develop REST APIs. Jersey is the default implementation. This starter project provides support to build JAX-RS-based REST APIs.

What other starters are there?

Starter	Description
spring-boot-starterwebsocket	HTTP is stateless. WebSockets allow you to maintain a connection between the server and the browser. This starter project provides support for Spring WebSockets.
spring-boot-starter-aop	This provides support for Aspect Oriented Programming. It also provides support for AspectJ for advanced aspect-oriented programming.
spring-boot-starter-amqp	With RabbitMQ as the default, this starter project provides message passing with AMQP.
spring-boot-starter-security	This starter project enables auto-configuration for Spring Security.
spring-boot-starter-data-jpa	This provides support for Spring Data JPA. Its default implementation is Hibernate.
spring-boot-starter	This is a base starter for Spring Boot applications. It provides support for auto-configuration and logging.
spring-boot-starter-batch	This provides support to develop batch applications using Spring Batch.
spring-boot-starter-cache	This is the basic support for caching using Spring Framework.
spring-boot-starter-datarest	This is the support to expose REST services using Spring Data REST.

spring-boot-starter-web

- This will add Spring MVC capabilities to Spring Boot
- Helps you build Spring MVC-based web applications or RESTful applications. It uses Tomcat as the default embedded servlet container.

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

spring-boot-starter-test

- The **spring-boot-starter-test** dependency provides the following test frameworks needed for unit testing:
  - **JUnit**: Basic unit test framework
  - **Mockito**: For mocking
  - **Hamcrest**, **AssertJ**: For readable asserts
  - **Spring Test**: A unit testing framework for spring-context based applications

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

spring-boot-maven-plugin

- When we build applications using Spring Boot, there are a couple of situations that are possible:
  - We would want to run the applications in place without building a JAR or a WAR
  - We would want to build a JAR and a WAR for later deployment
- The **spring-boot-maven-plugin** dependency provides capabilities for both of the preceding situations. The following snippet shows how we can configure spring-boot-maven-plugin in an application:

```
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
```

Create a Spring Hello Web project

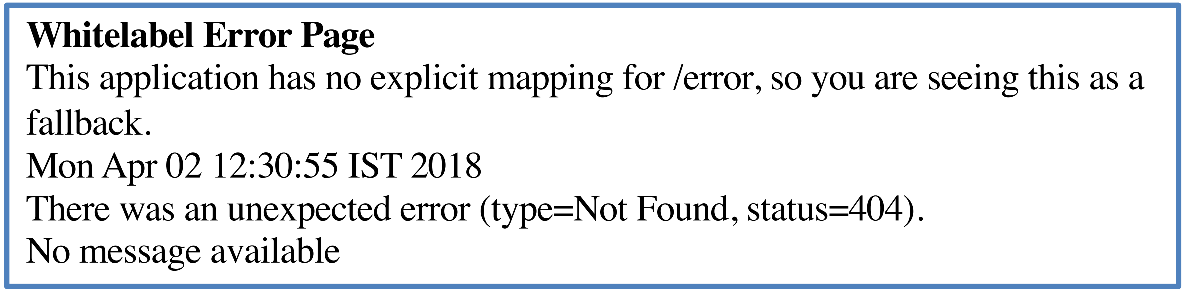
- Create a Spring hello web project
- Let’s understand what happens when

What happens when we execute

- Tomcat server is launched on port 8080 - Tomcat started on port(s): 8080 (http).
- DispatcherServlet is configured. This means that Spring MVC Framework is ready to accept requests--Mapping servlet: 'dispatcherServlet' to [/].
- Four filters are enabled by default
  - characterEncodingFilter
  - hiddenHttpMethodFilter,
  - httpPutFormContentFilter
  - requestContextFilter --
- The default error page is configured—
  - Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>>
  - org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet .http.HttpServletRequest)
- WebJars are autoconfigured. WebJars enable dependency management for static dependencies such as Bootstrap and query--Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]

What happens when we execute

- If we now open a browser and go to http://localhost:8080/ you will notice the default white label error page.



Whitelabel Error Page
This application has no explicit mapping for /error, so you are seeing this as a fallback.
Mon Apr 02 12:30:55 IST 2018
There was an unexpected error (type=Not Found, status=404).
No message available

Auto wired by Web starter

- when we add a dependency in spring-boot-starter-web, the following beans are auto-configured:
  - **basicErrorController, handlerExceptionResolver:** It is the basic exception handling. It shows a default error page when an exception occurs.
  - **beanNameHandlerMapping:** It is used to resolve paths to a handler (controller).
  - **characterEncodingFilter:** It provides default character encoding UTF-8.
  - **dispatcherServlet:** It is the front controller in Spring MVC applications.
  - **jacksonObjectMapper:** It translates objects to JSON and JSON to objects in REST services.
  - **messageConverters:** It is the default message converters to convert from objects into XML or JSON and vice versa.
  - **multipartResolver:** It provides support to upload files in web applications.
  - **mvcValidator:** It supports validation of HTTP requests.
  - **viewResolver:** It resolves a logical view name to a physical view.
  - **propertySourcesPlaceholderConfigurer:** It supports the externalization of application configuration.
  - **requestContextFilter:** It defaults the filter for requests.
  - **restTemplateBuilder:** It is used to make calls to REST services.
  - **tomcatEmbeddedServletContainerFactory:** Tomcat is the default embedded servlet container for Spring Boot-based web applications.

REST

- **Representational State Transfer** (**REST**) is basically an architectural style for the web. REST specifies a set of constraints. These constraints ensure that clients (service consumers and browsers) can interact with servers in flexible ways.
- **Terminology:**
  - **Server**: Service provider. Exposes services which can be consumed by clients.
  - **Client**: Service consumer. Could be a browser or another system.
  - **Resource**: Any information can be a resource: a person, an image, a video, or a product you want to sell.
  - **Representation**: A specific way a resource can be represented. For example, the product resource can be represented using JSON, XML, or HTML. Different clients might request different representations of the resource.

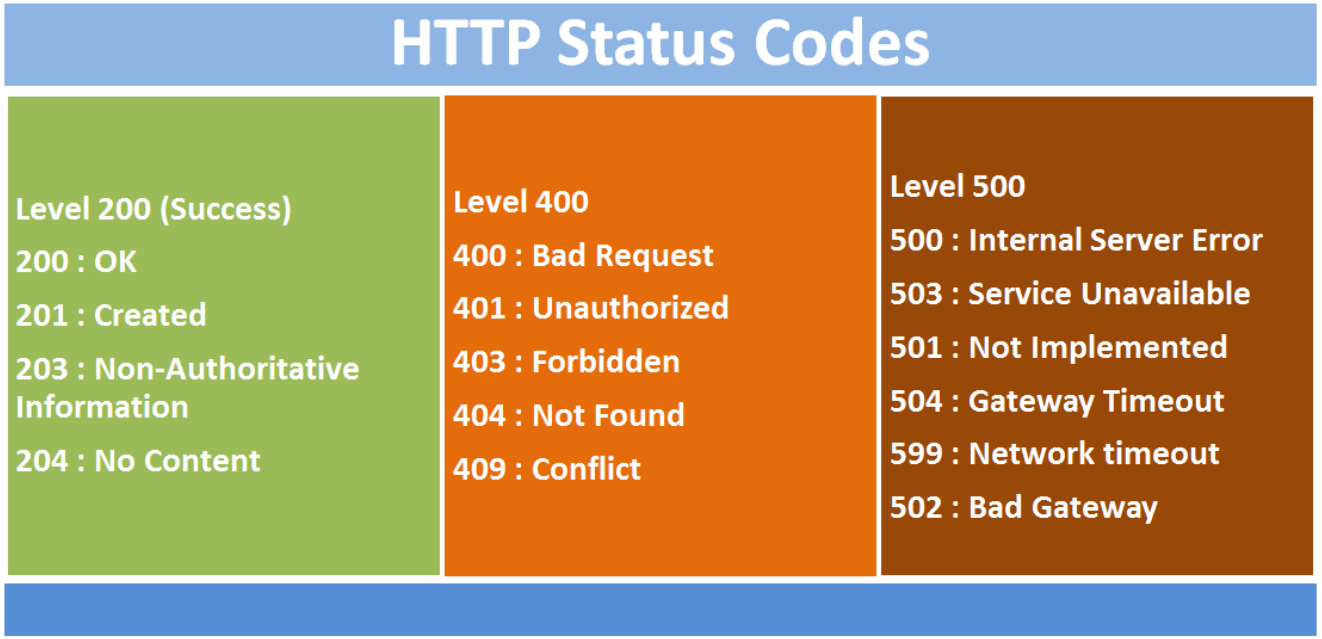
REST Constraints

- **Client-Server**: There should be a server (service provider) and a client (service consumer). This enables loose coupling and independent evolution of the server and client as new technologies emerge.
- **Stateless**: Each service should be stateless. Subsequent requests should not depend on some data from a previous request being temporarily stored. Messages should be self-descriptive.
- **Uniform interface**: Each resource has a resource identifier. In the case of web services, we use this URI example: /users/Jack/Todos/1. In this, URI Jack is the name of the user. 1 is the ID of the Todo we would want to retrieve.
- **Cacheable**: The service response should be cacheable. Each response should indicate whether it is cacheable.
- **Layered system**: The consumer of the service should not assume a direct connection to the service provider. Since requests can be cached, the client might be getting the cached response from a middle layer.
- **Manipulation of resources through representations**: A resource can have multiple representations. It should be possible to modify the resource through a message with any of these representations.
- **Hypermedia as the engine of application state** (**HATEOAS**): The consumer of a RESTful application should know about only one fixed service URL. All subsequent resources should be discoverable from the links included in the resource representations.

Http status codes

- HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:
  - informational responses
  - successful responses
  - redirects
  - client errors
  - servers errors.
- Go to: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

HTTP Status codes



HTTP Status Codes

Level 200 (Success)

200 : OK

201 : Created

203 : Non-Authoritative Information

204 : No Content

Level 400

400 : Bad Request

401 : Unauthorized

403 : Forbidden

404 : Not Found

409 : Conflict

Level 500

500 : Internal Server Error

503 : Service Unavailable

501 : Not Implemented

504 : Gateway Timeout

599 : Network timeout

502 : Bad Gateway

HTTP-REST Vocabulary

HTTP Methods supported by REST:

- GET – Requests a resource at the request URL
  - Should <u>not</u> contain a request body, as it will be discarded.
  - <u>May</u> be cached locally or on the server.
  - May produce a resource, but should not modify on it.
- POST – Submits information to the service for processing
  - Should typically return the new or modified resource.
- PUT – Add a new resource at the request URL
- DELETE – Removes the resource at the request URL
- OPTIONS – Indicates which methods are supported
- HEAD – Returns meta information about the request URL

Hello Rest

- Let's start with creating a simple REST service returning a welcome message
- creating a simple REST Controller method returning a string:

```
@RestController
public class HelloController {
@GetMapping("hello")
public String sayHello()
{
return "Hello Web!";
}
}
```

Hello Rest Cont..

- @RestController: The @RestController annotation provides a combination of @ResponseBody and @Controller annotations. This is typically used to create REST Controllers.
- @GetMapping("welcome"): @GetMapping is a shortcut for @RequestMapping(method = RequestMethod.GET). This annotation is a readable alternative. The method with this annotation would handle a Get request to the welcome URI.

```
@RestController
public class HelloController {
@GetMapping("hello")
public String sayHello()
{
return "Hello Web!";
}
}
```

Hello Rest - Execute

- Due to the latest changes in Spring boot 2.1 it's needs the following line to be added to application.properties to see mappings of URL

```
logging.level.org.springframework.web=TRACE
```

- When you execute this Spring application you will notice :

```
2019-01-02 05:18:56.637 TRACE 8089 --- [
main] s.w.s.m.m.a.RequestMappingHandlerMapping :
         c.c.f.c.HelloController:
         {GET /hello}: sayHello()
```

- Now if you open a browser and browse to http://localhost:8080/hello

```
Hello Web!
```

Hello with object

- Let’s add a new mapping and return a HelloWeb Object

```
@RestController
public class HelloController {
@GetMapping("/hello")
public String sayHello() {
return "Hello Web!";
}

@Autowired
HelloWeb helloWeb

@GetMapping("/helloweb")
public HelloWeb helloWithObject() {
helloWeb.setMessage("Hello Web from Bean");
return helloWeb;
}
}
```

ello with object cont..

- HelloWeb Class

```
@Component
public class HelloWeb {


private String message;


public String getMessage() {
return message;
}


public void setMessage(String message) {
this.message = message;
}
}
```

Output:

- Open a browser and go to : http://localhost:8080/helloweb

```
Content-Type →application/json;charset=UTF-8
Date →Mon, 02 Apr 2018 07:43:38 GMT
Transfer-Encoding →chunked
```

```
{
   "message": "Hello Web from Bean"
}
```

- As observed with out any intervention Spring Web starter is allowing us to create JSON responses from Java Objects
- Again, it's the magic of Spring Boot auto-configuration. If Jackson is on the classpath of an application, instances of the default object to JSON (and vice versa) converters are auto-configured by Spring Boot.

Hello with path param

- HelloController Class extended to have one more method and get mapping.

```
private static final String helloWorldTemplate = "Hello %s!";


@GetMapping("/helloweb/name/{name}")
public HelloWeb helloWithObject(@PathVariable String name) {
HelloWeb helloWeb = new HelloWeb();
helloWeb.setMessage(String.format(helloWorldTemplate, name));
return helloWeb;
}

@GetMapping("/helloweb/name")
public HelloWeb helloWithObject(@RequestParam String name) {
HelloWeb helloWeb = new HelloWeb();
helloWeb.setMessage(String.format(helloWorldTemplate, name));
return helloWeb;
}
```

Output:

- Open a browser and go to : http://localhost:8080/helloweb/name/john

```
Content-Type →application/json;charset=UTF-8
Date →Mon, 02 Apr 2018 07:43:38 GMT
Transfer-Encoding →chunked
```

```
{
  "message": "Hello john!"
}
```

- @GetMapping("/helloweb/name/{name}"): {name} indicates that this value will be the variable. We can have multiple variable templates in a URI.
- welcomeWithParameter(@PathVariable String name): @PathVariable ensures that the variable value from the URI is bound to the variable name.
- String.format(helloWorldTemplate, name): A simple string format to replace %s in the template with the name.

Unit Testing using MockMVC:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class HelloControllerTest {
  @Autowired
      private MockMvc mvc;

  @Test
    public void welcomeWithParameter() throws Exception {

mvc.perform(
      MockMvcRequestBuilders.get("/helloweb/name/john")
     .accept(MediaType.APPLICATION_JSON))
     .andExpect(status().isOk())
     .andExpect(
     content().string(containsString("Hello john!")));
    }
}
```

Unit Testing using MockMVC:

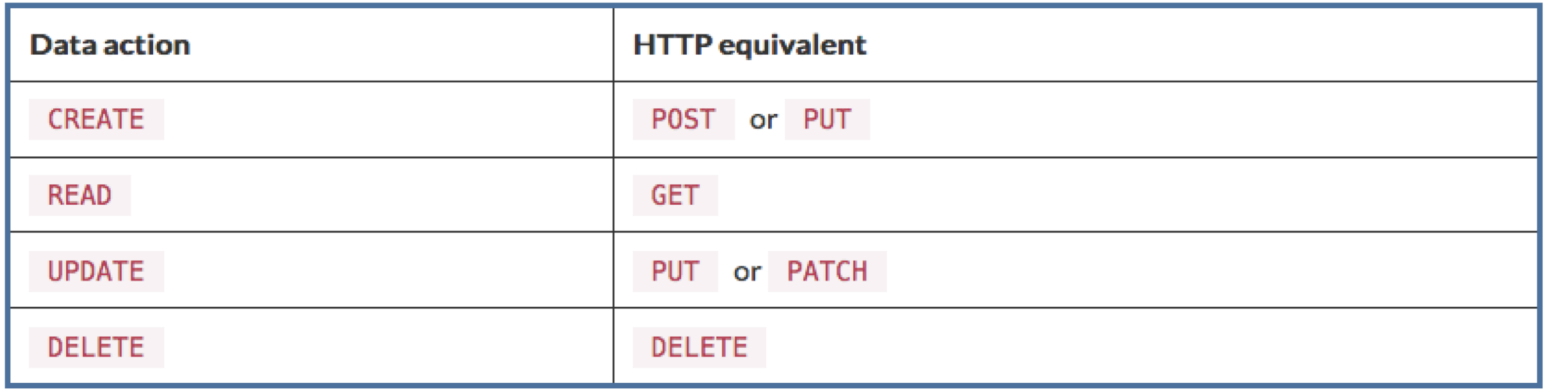
- we will launch up a Mock MVC instance with HelloController. A few quick things to note are as follows:
  - @RunWith(**SpringRunner.class**): SpringRunner is a shortcut to the SpringJUnit4ClassRunner annotation. This launches up a simple Spring context for unit testing.
  - @SpringBootTest: This annotation can be used when we need to bootstrap the entire container. The annotation works by creating the ApplicationContext that will be utilized in our tests.
  - @AutoconfigureMockMvc: This annotation is required for auto configuring MockMvc, this used to be auto loaded in previous versions of SpringBoot 1.x, not any more.
  - @Autowired private MockMvc mvc: Autowires the MockMvc bean that can be used to make requests.
  - mvc.perform(MockMvcRequestBuilders.get("/hello/name/john").accept(MediaType.APPLICATION_JSON)): Performs a request to /welcome with the Accept header value application/json.
  - andExpect(status().isOk()): Expects that the status of the response is 200 (success).
  - andExpect(content().string(containsString("Hello john!"))): Expects that the content of the response is equal to "Hello john!".

Todo REST Service

- We will focus on creating REST services for a basic Todo management system. We will create services for the following:
  - Retrieving a list of Todos for a given user
  - Retrieving details for a specific Todo
  - Creating a Todo for a user
- To keep things simple, this service does not talk to the database. It maintains an in-memory array list of Todos. This list is initialized using a static initializer.
- We are exposing a couple of simple retrieve methods and a method to add a to-do.

Generic interaction semantics for REST resources

- HTTP specifies methods or actions for the resources. The most commonly used HTTP methods or actions are POST, GET, PUT, and DELETE. This clearly simplifies the REST API design and makes it more readable.
- In a RESTful system, we can easily map our **CRUD** actions on the resources to the appropriate HTTP methods such as POST, GET, PUT, and DELETE

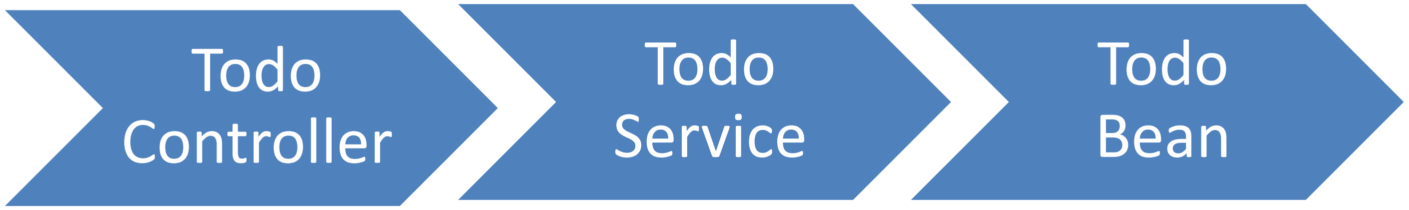


Data action	HTTP equivalent
`CREATE`	`POST` or `PUT`
`READ`	`GET`
`UPDATE`	`PUT` or `PATCH`
`DELETE`	`DELETE`

Todos REST Mapping

- Let's quickly map the services that we want to create to the appropriate request methods:
  - **Retrieving a list of Todos for a given user**: This is READ. We will use GET. We will use a URI: /users/{name}/Todos. One more good practice is to use plurals for static things in the URI: users, Todo, and so on. This results in more readable URIs.
  - **Retrieving details for a specific Todo**: Again, we will use GET. We will use a URI /users/{name}/Todos/{id}. You can see that this is consistent with the earlier URI that we decided for the list of Todos.
  - **Creating a Todo for a user**: For the create operation, the suggested HTTP Request method is POST. To create a new Todo, we will post to URI /users/{name}/Todos.

Todo coding



Todo Bean

- We have a created a simple Todo bean with the ID, the name of user, the description of the Todo, the Todo target date, and an indicator for the completion status. We added a constructor and getters for all fields.

```
public class Todo {

private int id;
private String user;
private String description;
private Date targetDate;
private boolean isDone;

public Todo(int id, String user, String desc, Date targetDate, boolean isDone)
{
this.id = id; this.user = user;
this.desc = desc; this.targetDate = targetDate; this.isDone = isDone;
}
//all getters
```

Todo Service

- **We have a created a simple Todo bean with the ID, the name of user, the description of the Todo, the Todo target date, and an indicator for the completion status. We added a constructor and getters for all fields.**

```
@Service
public class TodoService {
private static List<Todo> Todos = new ArrayList<Todo>();
private static int TodoCount = 3;

static {
Todos.add(new Todo(1, "Jack", "Learn Spring Boot", new Date(), false));
Todos.add(new Todo(2, "Jack", "Learn JPA", new Date(), false));
Todos.add(new Todo(3, "Jill", "Learn AWS Lambda", new Date(), false));
}

public List<Todo> retrieveTodos(String user) {
List<Todo> filteredTodos = new ArrayList<Todo>();
for (Todo Todo : Todos) {
if (Todo.getUser().equals(user))
filteredTodos.add(Todo);
}
return filteredTodos;
}
Contd..
```

Todo Service Contd..

```
//for adding Todo.
public Todo addTodo(String name, String desc, Date targetDate,
boolean isDone) {
Todo Todo = new Todo(TodoCount++, name, desc, targetDate, isDone);
Todos.add(Todo);
return Todo;
}

//retrieve based on Id.
public Todo retrieveTodo(int id) {
for (Todo Todo : Todos) {
if (Todo.getId() == id)
return Todo;
}
return null;
}
}
```

@Service

- The @Service annotation is also a specialization of the component annotation. It doesn’t currently provide any additional behavior over the @Component annotation, but it’s a good idea to use @Service over @Component in service-layer classes because **it specifies intent better**. Additionally, tool support and additional behavior might rely on it in the future.

Retrieving a Todo List

- We will create a new RestController annotation called TodoController.

```
@RestController
public class TodoController {


@Autowired
private TodoService TodoService;


@GetMapping("/users/{name}/Todos")
public List<Todo> retrieveTodos(@PathVariable String
name) {
return TodoService.retrieveTodos(name);
}
}
```

What we did?

- We are autowiring the Todo service using the @Autowired annotation
- We use the @GetMapping annotation to map the Get request for the "/users/{name}/Todos" URI to the retrieveTodos method

Retrieving a Todo List by ID

- The URI mapped is /users/{name}/Todos/{id}
- We have two path variables defined for name and id

```
@RestController
public class TodoController {


@Autowired
private TodoService TodoService;

@GetMapping(path = "/users/{name}/Todos/{id}")
    public Todo retrieveTodo(@PathVariable String name,
@PathVariable
    int id) {
        return TodoService.retrieveTodo(id);
    }
}
```

Let’s Verify Retrieval by name

- Go to http://localhost:8080/users/Jack/Todos

```
[
    {
        "id": 1,
        "user": "Jack",
        "desc": "Learn Spring Boot",
        "targetDate": 1522679821841,
        "done": false
    },
    {
        "id": 2,
        "user": "Jack",
        "desc": "Learn JPA",
        "targetDate": 1522679821841,
        "done": false
    }
]
```

Let’s Verify Retrieval by ID of a User

- Go to http://localhost:8080/users/Jack/Todos/1

```
{
        "id": 1,
        "user": "Jack",
        "desc": "Learn Spring Boot",
        "targetDate": 1522679821841,
        "done": false
}
```

- Go to http://localhost:8080/users/Jack/Todos/300

```
{
        "id": 1,
        "user": "Jack",
        "desc": "Learn Spring Boot",
        "targetDate": 1522679821841,
        "done": false
}
```

Adding A Todo

- We will now add the method to create a new Todo. The HTTP method to be used for creation is Post. We will post to a "/users/{name}/Todos" URI:

```
@PostMapping("/users/{name}/Todos")
ResponseEntity<?> add(@PathVariable String name, @RequestBody Todo Todo) {

Todo createdTodo = TodoService.addTodo(name, Todo.getDesc(),
Todo.getTargetDate(), Todo.isDone());
if (createdTodo == null) {
return ResponseEntity.noContent().build();
}

URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
.buildAndExpand(createdTodo.getId()).toUri();
return ResponseEntity.created(location).build();
}
```

Adding A Todo Contd..

- @PostMapping("/users/{name}/Todos"): @PostMapping annotations map the add() method to the HTTP Request with a POST method.
- ResponseEntity<?> add(@PathVariable String name, @RequestBody Todo Todo): An HTTP post request should ideally return the URI to the created resources. We use ResourceEntity to do this. @RequestBody binds the body of the request directly to the bean.
- ResponseEntity.noContent().build(): Used to return that the creation of the resource failed. You can even throw an expection.
- ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(createdTodo.getId()).toUri(): Forms the URI for the created resource that can be returned in the response.
- ResponseEntity.created(location).build(): Returns a status of 201(CREATED) with a link to the resource created.

Let’s Verify Retrieval by ID of a User

- POST http://localhost:8080/users/Jack/Todos

```
{

          "user": "Jack",
          "desc": "Learn Spring Boot 2",
          "done": false
}
```

- Response

```
Content-Length →0
Date →Mon, 02 Apr 2018 16:25:16 GMT
Location →http://localhost:8080/users/Jack/Todos/4
```

Exception Handling

- Exception handling is one of the important parts of developing web services. When something goes wrong, we would want to return a good description of what went wrong to the service consumer.
- Spring Boot provides good default exception handling. We will start with looking at the default exception handling features provided by Spring Boot before moving on to customizing them.
- Let's send a GET request to http://localhost:8080/non-existing-resource

```
{
    "timestamp": 1522690043731,
    "status": 404,
    "error": "Not Found",
    "message": "No message available",
    "path": "/non-existing-resource"
}
```

Exception Handling - default

- The response header has an HTTP status of 404 - Resource Not Found
- Spring Boot returns a valid JSON message as a response with the message stating that the resource is not found

```
Content-Type →application/json;charset=UTF-8
Date →Mon, 02 Apr 2018 17:27:23 GMT
Transfer-Encoding →chunked
```

```
{
    "timestamp": 1522690043731,
    "status": 404,
    "error": "Not Found",
    "message": "No message available",
    "path": "/non-existing-resource"
}
```

Resource Throwing an Exception

- Let's create a resource that throws an exception, and send a GET request to it in order to understand how the application reacts to runtime exceptions.
- We are creating a GET service with the URI /users/dummy-service.
- The service throws a RuntimeException. We chose RuntimeException to be able to create the exception easily. We can easily replace it with a custom exception; if needed.

```
@GetMapping(path = "/users/dummy-service")
    public Todo errorService() {
      throw new RuntimeException("Some Exception Occured");
    }
```

Exception Handling - Exception

- The response header has an HTTP status of 500; Internal server error
- Spring Boot also returns the message with which the exception is thrown

Connection →close
Content-Type →application/json;charset=UTF-8
Date →Mon, 02 Apr 2018 17:42:22 GMT
Transfer-Encoding →chunked

```
{
    "timestamp": 1522690942014,
    "status": 500,
    "error": "Internal Server Error",
    "exception": "java.lang.RuntimeException",
    "message": "Some Exception Occured",
    "path": "/users/dummy-service"
}
```

Throwing a Custom Exception

- It's a very simple piece of code that defines TodoNotFoundException.
- Now let's enhance our TodoController class to throw TodoNotFoundException when a Todo with a given ID is not found:

```
public class TodoNotFoundException extends
RuntimeException {
    public TodoNotFoundException(String msg) {
      super(msg);
    }
  }
```

Throwing a Custom Exception

- It's a very simple piece of code that defines TodoNotFoundException.
- Now let's enhance our TodoController class to throw TodoNotFoundException when a Todo with a given ID is not found:

```
public class TodoNotFoundException extends RuntimeException {
    public TodoNotFoundException(String msg) {
      super(msg);
    }
  }
```

```
@GetMapping("/users/{name}/Todos/{id}")
public Todo retrieveTodo(@PathVariable String name, @PathVariable int id) {
Todo Todo = TodoService.retrieveTodo(id);
if (Todo == null) {
throw new TodoNotFoundException("Todo Not Found");
}
return Todo;
}
```

Throwing a Custom Exception

- If TodoService returns a null Todo, we throw; TodoNotFoundException.
- When we execute the service with a GET request to a nonexistent: Todo(http://localhost:8080/users/Jack/Todos/300), we get the response shown in the following code snippet:

```
{
    "timestamp": 1522692318250,
    "status": 500,
    "error": "Internal Server Error",
    "exception": "com.verizon.apps.exceptions.TodoNotFoundException",
    "message": "Todo Not Found",
    "path": "/users/Jack/Todos/300"
}
```

- As we can see, a clear exception response is sent back to the service consumer. However, there is one thing that can be improved further--the response status. When a resource is not found, it is recommended that you return a 404 - Resource Not Found status.

Customizing Exception Response

- Let's create a bean to define the structure of our custom exception message:

```
public class ExceptionResponse {
private Date timestamp = new Date();
private String message;
private String details;

public ExceptionResponse(String message,
String details) {
super();
this.message = message;
this.details = details;
}
}
```

Customizing Exception Response

- When; TodoNotFoundException is thrown, we would want to return a response using the ExceptionResponse bean. The following code shows how we can create a global exception handling for TodoNotFoundException.class

```
@ControllerAdvice
@RestController
public class RestResponseEntityExceptionHandler extends ResponseEntityExceptionHandler
{

@ExceptionHandler(TodoNotFoundException.class)
public final ResponseEntity<ExceptionResponse> TodoNotFound(TodoNotFoundException ex)
{
ExceptionResponse exceptionResponse = new ExceptionResponse(ex.getMessage(),
"Any details you would want to add");
return new ResponseEntity<ExceptionResponse>(exceptionResponse, new HttpHeaders(),
HttpStatus.NOT_FOUND);
}
}
```

Customizing Exception Response

- *RestResponseEntityExceptionHandler extends ResponseEntityExceptionHandler:* We are extending ResponseEntityExceptionHandler, which the base class is provided by Spring MVC for centralized exception handling ControllerAdvice classes.
- @ExceptionHandler(TodoNotFoundException.class): This defines that the method to follow will handle the specific exception TodoNotFoundException.class. Any other exceptions for which custom exception handling is not defined will follow the default exception handling provided by Spring Boot.
- ExceptionResponse exceptionResponse = new ExceptionResponse(ex.getMessage(), "Any details you would want to add"): This creates a custom exception response.
- new ResponseEntity<ExceptionResponse>(exceptionResponse,new HttpHeaders(), HttpStatus.NOT_FOUND): This is the definition to return a 404 Resource Not Found response with the custom exception defined earlier.
- When we execute the service with a GET request to a nonexistent; Todo(http://localhost:8080/users/Jack/Todos/300), we get the following response:

Customizing Exception Response

- When we execute the service with a GET request to a nonexistent; Todo(http://localhost:8080/users/Jack/Todos/300), we get the following response:

```
{
    "timestamp": 1522692929862,
    "message": "Todo Not Found",
    "details": "Any details you would want to add"
}
```

- If you want to create a generic exception message for all exceptions, we can add a method to RestResponseEntityExceptionHandler with the @ExceptionHandler(Exception.class) annotation.

```
@ExceptionHandler(Exception.class)
public final ResponseEntity<ExceptionResponse>
TodoNotFound(TodoNotFoundException ex) {
}
```

Documenting REST Services

- Before a service provider can consume a service, they need a service contract. A service contract defines all the; details about a service:
  - How can I call a service? What is the URI of the service?
  - What should be the request format?
  - What kind of response should I expect?
- There are multiple options to define a service contract for RESTful services. The most popular one in the last couple of years is **Swagger**.
- Swagger specification creates the RESTful contract for your API, detailing all of its resources and operations in a human and machine readable format for easy development, discovery, and integration.

Developer Tools

- Spring Boot provides tools that can improve the experience of developing Spring Boot applications.
- Spring Boot developer tools, by default, disables the caching of view templates and static files. This enables a developer to see the changes as soon as they make them.
- Another important feature is the automatic restart when any file in the classpath changes. So, the application automatically restarts in the following scenarios:
  - When we make a change to a controller or a service class
  - When we make a change to the property file

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-devtools</artifactId>
<optional>true</optional>
</dependency>
```

Developer Tools

- The advantages of Spring Boot developer tools are as follows:
  - The developer does not need to stop and start the application each time. The application is automatically restarted as soon as there is a change.
  - The restart feature in Spring Boot developer tools is intelligent. It only reloads the actively developed classes. It does not reload the third-party JARs (using two different class-loaders). Thereby, the restart when something in the application changes is much faster compared to cold-starting an application.

Spring Boot Actuator

- When an application is deployed into production:
  - We want to know immediately if some service goes down or is very slow
  - We want to know immediately if any of the servers does not have sufficient free space or memory
- This is called **application monitoring**.
- **Spring Boot Actuator** provides a number of production-ready monitoring features.

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

Spring Boot Actuator

- As soon as the actuator is added to an application, it enables a number of endpoints. When we start the application, we see a number of added new mappings.
- Go to : http://localhost:8080/actuator

```
management.endpoints.web.exposure.include=*
```

- You will see lot of details and let's explore, but before that Security has to be disabled.

Spring Boot Actuator

```
{
    "_links": {
        "self": {
            "href": "http://localhost:8080/actuator",
            "templated": false
        },
        "archaius": {
            "href": "http://localhost:8080/actuator/archaius",
            "templated": false
        },
        "auditevents": {
            "href": "http://localhost:8080/actuator/auditevents",
            "templated": false
        },
        "beans": {
            "href": "http://localhost:8080/actuator/beans",
            "templated": false
        },
        "caches-cache": {
            "href": "http://localhost:8080/actuator/caches/{cache}",
            "templated": true
        },
        "caches": {
            "href": "http://localhost:8080/actuator/caches",
            "templated": false
        },
        "health": {
            "href": "http://localhost:8080/actuator/health",
            "templated": false
        },
        "health-component-instance": {
            "href": "http://localhost:8080/actuator/health/{component}/{instance}",
            "templated": true
        },
        "conditions": {
            "href": "http://localhost:8080/actuator/conditions",
            "templated": false
        },
        "configprops": {
            "href": "http://localhost:8080/actuator/configprops",
            "templated": false
        },
```

Spring Boot Actuator

```
 "env": {
    "href": "http://localhost:8080/actuator/env",
    "templated": false
},
"env-toMatch": {
    "href": "http://localhost:8080/actuator/env/{toMatch}",
    "templated": true
},
"info": {
    "href": "http://localhost:8080/actuator/info",
    "templated": false
},
"loggers-name": {
    "href": "http://localhost:8080/actuator/loggers/{name}",
    "templated": true
},
"loggers": {
    "href": "http://localhost:8080/actuator/loggers",
    "templated": false
},
"heapdump": {
    "href": "http://localhost:8080/actuator/heapdump",
    "templated": false
},
"threaddump": {
    "href": "http://localhost:8080/actuator/threaddump",
    "templated": false
},
"metrics-requiredMetricName": {
    "href": "http://localhost:8080/actuator/metrics/{requiredMetricName}",
    "templated": true
},
"metrics": {
    "href": "http://localhost:8080/actuator/metrics",
    "templated": false
},
"scheduledtasks": {
    "href": "http://localhost:8080/actuator/scheduledtasks",
    "templated": false
},
```

Spring Boot Actuator

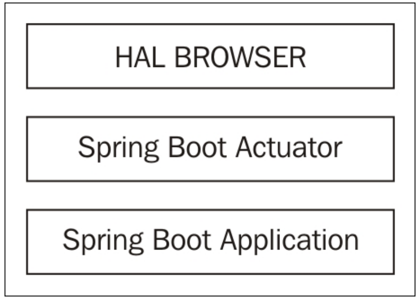
```
"httptrace": {
            "href": "http://localhost:8080/actuator/httptrace",
            "templated": false
        },
        "mappings": {
            "href": "http://localhost:8080/actuator/mappings",
            "templated": false
        },
        "refresh": {
            "href": "http://localhost:8080/actuator/refresh",
            "templated": false
        },
        "features": {
            "href": "http://localhost:8080/actuator/features",
            "templated": false
        },
        "service-registry": {
            "href": "http://localhost:8080/actuator/service-registry",
            "templated": false
        }
    }
}
```

HAL Browser

- A number of these endpoints expose a lot of data. To be able to visualize the information better, we will add an **HAL Browser** to our application:

```
<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-rest-hal-browser</artifactId>
</dependency>
```

- Spring Boot Actuator exposes REST APIs around all the data captured from the Spring Boot application and environment. The HAL Browser enables visual representation around the Spring Boot Actuator API:



HAL Browser

- Go to : http://localhost:8080/browser/index.html

Or

- http://localhost:8080/browser/index.html#/actuator

Spring Shortcut annotations

- Spring currently supports five types of inbuilt annotations for handling different types of incoming HTTP request methods which are *GET, POST, PUT, DELETE* and *PATCH*. These annotations are:
- *@GetMapping*
- *@PostMapping*
- *@PutMapping*
- *@DeleteMapping*
- *@PatchMapping*

Spring JPA

- The Java Persistence API (JPA) is an Object Relational Mapping (ORM) framework that’s part of the Java EE platform.
- JPA simplifies the implementation of the data access layer by letting developers work with an object oriented API instead of writing SQL queries by hand. The most popular JPA implementations are Hibernate, EclipseLink, and OpenJPA.
- The Spring framework provides a Spring ORM module to integrate easily with ORM frameworks. You can also use Spring's declarative transaction management capabilities with JPA. In addition to the Spring ORM module, the Spring data portfolio project provides a consistent, Spring-based programming model for data access to relational and NoSQL datastores.
- Spring Data integrates with most of the popular data access technologies, including JPA, MongoDB, Redis, Cassandra, Solr, ElasticSearch, etc.

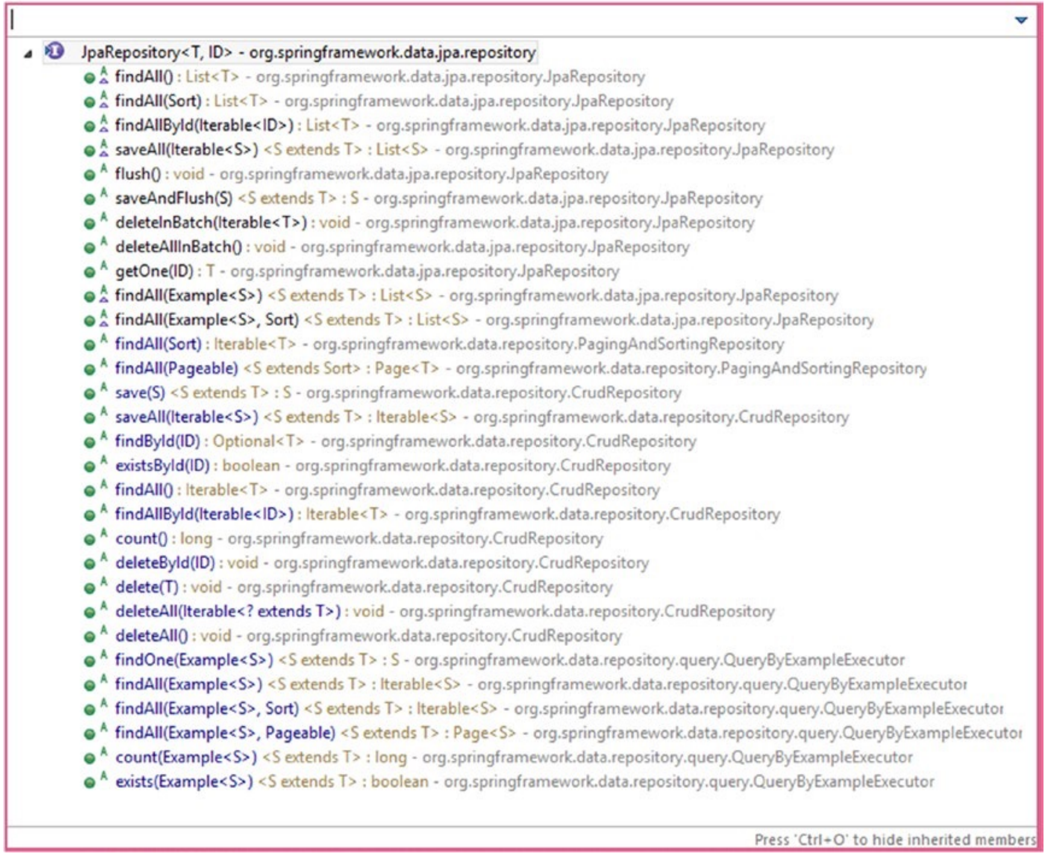
Spring JPA Features

- Sophisticated support to build repositories based on Spring and JPA
- Support for Querydsl predicates and thus type-safe JPA queries
- Transparent auditing of domain class
- Pagination support, dynamic query execution, ability to integrate custom data access code
- Validation of @Query annotated queries at bootstrap time
- Support for XML based entity mapping
- JavaConfig based repository configuration by introducing @EnableJpaRepositories.

Spring Data JPA

- Spring Data is an umbrella project that provides data access support for most of the popular data access technologies—including JPA, MongoDB, Redis, Cassandra, Solr, and ElasticSearch—in a consistent programming model.
- Spring Data JPA is one of the modules for working with relational databases using JPA.
- At times, you may need to implement the data management applications to store, edit, and delete data. For those applications, you just need to implement CRUD (Create, Read, Update, Delete) operations for entities. Instead of implementing the same CRUD operations again and again or rolling out your own generic CRUD DAO implementation.
- Spring Data provides various repository abstractions, such as CrudRepository, PagingAndSortingRepository, JpaRepository, etc. They provide out-of-the-box support for CRUD operations, as well as pagination and sorting.

Spring Data JPA



```
JpaRepository<T, ID> - org.springframework.data.jpa.repository
    findAll() : List<T> - org.springframework.data.jpa.repository.JpaRepository
    findAll(Sort) : List<T> - org.springframework.data.jpa.repository.JpaRepository
    findAllById(Iterable<ID>) : List<T> - org.springframework.data.jpa.repository.JpaRepository
    saveAll(Iterable<S>) <S extends T> : List<S> - org.springframework.data.jpa.repository.JpaRepository
    flush() : void - org.springframework.data.jpa.repository.JpaRepository
    saveAndFlush(S) <S extends T> : S - org.springframework.data.jpa.repository.JpaRepository
    deleteInBatch(Iterable<T>) : void - org.springframework.data.jpa.repository.JpaRepository
    deleteAllInBatch() : void - org.springframework.data.jpa.repository.JpaRepository
    getOne(ID) : T - org.springframework.data.jpa.repository.JpaRepository
    findAll(Example<S>) <S extends T> : List<S> - org.springframework.data.jpa.repository.JpaRepository
    findAll(Example<S>, Sort) <S extends T> : List<S> - org.springframework.data.jpa.repository.JpaRepository
    findAll(Sort) : Iterable<T> - org.springframework.data.repository.PagingAndSortingRepository
    findAll(Pageable) <S extends Sort> : Page<T> - org.springframework.data.repository.PagingAndSortingRepository
    save(S) <S extends T> : S - org.springframework.data.repository.CrudRepository
    saveAll(Iterable<S>) <S extends T> : Iterable<S> - org.springframework.data.repository.CrudRepository
    findById(ID) : Optional<T> - org.springframework.data.repository.CrudRepository
    existsById(ID) : boolean - org.springframework.data.repository.CrudRepository
    findAll() : Iterable<T> - org.springframework.data.repository.CrudRepository
    findAllById(Iterable<ID>) : Iterable<T> - org.springframework.data.repository.CrudRepository
    count() : long - org.springframework.data.repository.CrudRepository
    deleteById(ID) : void - org.springframework.data.repository.CrudRepository
    delete(T) : void - org.springframework.data.repository.CrudRepository
    deleteAll(Iterable<? extends T>) : void - org.springframework.data.repository.CrudRepository
    deleteAll() : void - org.springframework.data.repository.CrudRepository
    findOne(Example<S>) <S extends T> : S - org.springframework.data.repository.query.QueryByExampleExecutor
    findAll(Example<S>) <S extends T> : Iterable<S> - org.springframework.data.repository.query.QueryByExampleExecutor
    findAll(Example<S>, Sort) <S extends T> : Iterable<S> - org.springframework.data.repository.query.QueryByExampleExecutor
    findAll(Example<S>, Pageable) <S extends T> : Page<S> - org.springframework.data.repository.query.QueryByExampleExecutor
    count(Example<S>) <S extends T> : long - org.springframework.data.repository.query.QueryByExampleExecutor
    exists(Example<S>) <S extends T> : boolean - org.springframework.data.repository.query.QueryByExampleExecutor
```

Press 'Ctrl+O' to hide inherited members

Spring Data JPA

- JpaRepository provides several methods for CRUD operations, along with the following interesting methods:
  - long count();—Returns the total number of entities available.
  - boolean existsById(ID id);—Returns whether an entity with the given ID exists.
  - List<T> findAll(Sort sort);—Returns all entities sorted by the given options.
  - Page<T> findAll(Pageable pageable);—Returns a page of entities meeting the paging restriction provided in the Pageable object.
- Spring Data JPA not only provides CRUD operations out-of-the-box, but it also supports dynamic query generation based on the method names.
  - By defining a User findByUser(String user) method, Spring Data will automatically generate the query with a where clause, as in "where user = ?1".
  - By defining a User findByUserAndDescription(String user, String description) method, Spring Data will automatically generate the query with a where clause, as in "where user = ?1 and description=?2".
  - http://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

Spring Data JPA

- But sometimes you may not be able to express your criteria using method names or the method names look ugly. Spring Data provides flexibility to configure the query explicitly using the @Query annotation.

  -
    ```
    @Query("select u from User u where u.user=?1 and u.description=?2 and u.done=true")
        Todo findByUserAndDescription(String user, String description);
    ```

- You can also perform data update operations using @Modifying and @Query, as follows:

  -
    ```
    @Modifying @Query("update User u set u.done=:done")
        int updateDone(@Param("done") boolean done)
    ```

  Note that this example uses the named parameter :done instead of the positional parameter ?1.

Let’s use Spring Data JPA with H2 Database

- Add the dependencies needed for Spring Data JPA and H2 Database

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<scope>runtime</scope>
</dependency>
```

- Update application.properties file

```
spring.h2.console.enabled=true
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
```

Update the Todo Bean

- Update the TODO bean with @Entity and @

```
@Entity
@Component
@Table( name="TODO" )
public class Todo {

@Id
@GeneratedValue
private int id;
private String user;
@Size(min=5, max=15, message= "Description must be b/w 5 to 15 Characters!")
private String description;
private boolean done;
@Column( name ="TARGETDATE")
private Date targetDate;

//all getter setters
}
```

Create the Repository class

- representation around the Spring Boot Actuator API:

```
@Repository
public interface TodoRepository extends JpaRepository<Todo, Integer>{

List<Todo> findByUser(String user);

}
```

Create the Service class

-
  ```
  @Service
  public class TodoService {
  @Autowired
  TodoRepository TodoRepository;

  public Todo createTodo(Todo Todo) {
  return TodoRepository.save(Todo);
  }
  public List<Todo> getTodos() {
  return TodoRepository.findAll();
  }
  public List<Todo> getTodosByUser(String user) {
  return TodoRepository.findByUser(user);
  }
  public Optional<Todo> getTodoById(int id) {
  return TodoRepository.findById(id);
  }
  }
  ```

Update the Controller class

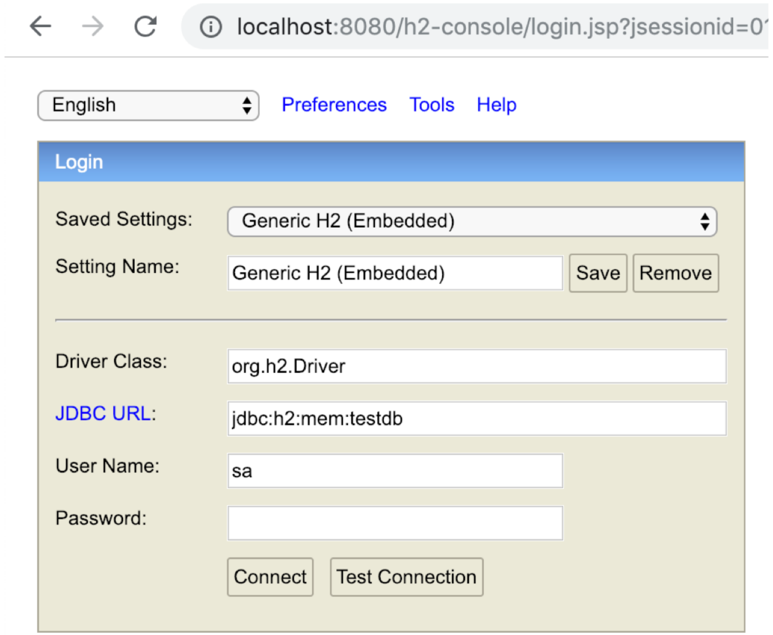
-
  ```
@RestController
public class TodoController {

@Autowired
TodoService TodoService;

@GetMapping("/Todo/{id}")
Optional<Todo> getTodoByID(@PathVariable int id) {
return TodoService.getTodoById(id);
}
@GetMapping("/Todos/{user}")
List<Todo> getTodosByUser(@PathVariable String user) {
return TodoService.getTodosByUser(user);
}
@GetMapping("/Todos")
List<Todo> getTodos()
{
return TodoService.getTodos();
}
@PostMapping("/Todo")
Todo createTodo(@Valid @RequestBody Todo Todo) {
return TodoService.createTodo(Todo);
}
}
```

Accessing the H2 Console

- Open http://localhost:8080/h2-console
- User jdbc:h2:mem:testdb which is the default H2 database URL



localhost:8080/h2-console/login.jsp?jsessionid=0

English Preferences Tools Help

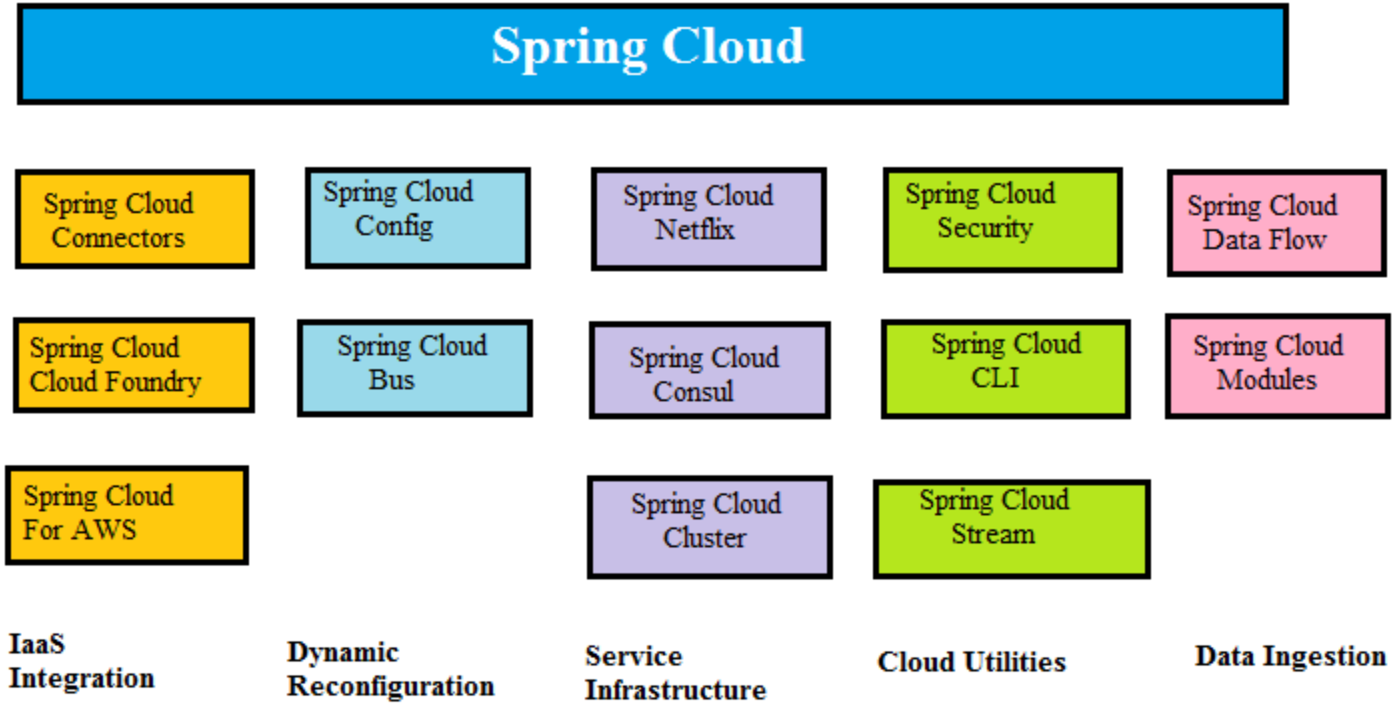
Login

Saved Settings:	Generic H2 (Embedded)
Setting Name:	Generic H2 (Embedded)
Driver Class:	org.h2.Driver
JDBC URL:	jdbc:h2:mem:testdb
User Name:	sa
Password:	

Spring Annotations

Go To: https://springframework.guru/spring-framework-annotations/

Spring cloud projects



Spring Cloud Kubernetes

- Spring Cloud Kubernetes provides implementations of well known Spring Cloud interfaces allowing developers to build and run Spring Cloud applications on Kubernetes.
- Features
  - Kubernetes awareness
  - DiscoveryClient implementation
  - PropertySource objects configured via ConfigMaps
  - Client side loadbalancing via Netflix Ribbon

THANK YOU!