SpringBoot

By

Praveen Naga

Blog: https://praveennagatech.blogspot.com

Facebook Group: https://www.facebook.com/groups/268426377837151

Github repo: https://github.com/praveennaga

Email: opraveennaga@gmail.com

Please make sure to refer my Spring Framework ebook
https://github.com/praveennaga/PraveenNaga-Tech-
Ebooks/blob/master/Praveen%20Naga_Spring%20Framework.pdf before proceeding
further on this SpringBoot ebook.

What is Spring Boot?

Spring Boot is basically an extension of the Spring framework which eliminates the
boilerplate configurations required for setting up a Spring application.

Features of Spring Boot

- ✓ Create stand-alone Spring applications with
  Embeded Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- ✓ Provide opinionated 'starter' dependencies to simplify your build configuration.
  Opinionated in the sense Framework choose configurations. It will configure
  basic setup for you based on some parameters.
- ✓ Automatically configure Spring and 3rd party libraries whenever possible
- ✓ Provide production-ready features such as metrics, health checks and
  externalized configuration
- ✓ Absolutely no code generation and no requirement for XML configuration
- ✓ It is highly dependent on the starter templates feature which is very powerful and
  works flawlessly.

Spring vs Spring Boot

Spring

- ✓ Dependency Injection Framework
- ✓ Manage lifecycle of java classes(beans)
- ✓ Boiler plate configuration(programmer writes a lot of code to do minimal task)
- ✓ Takes time to have a spring application up and running

Spring Framework	Spring Boot
More boilerplate code	Reduce boilerplate code
XML Configuration	Annotations

Benefits of Spring Boot

- ✓ Dependency Resolution
- ✓ Minimum Configuration
- ✓ Embedded server for testing

- ✓ Bean auto scan
- ✓ Health metrics
- ✓ Standarization for microservices
- ✓ Cloud support
- ✓ Adapt and support for 3rd party libraries

In one sentence, Spring Boot is (Spring Framework-XML Configuration) +Integrated server.

What SpringBoot is NOT!
- ✓ Zero code generation
- ✓ Neither an application server nor a webserver

Why is it "opinionated"?

It makes assumptions on what you need based on dependencies from the classpath. Convention over configuration - pre-configures Spring app by reasonable defaults, which can be overridden.

How does it work? How does it know what to configure?

Spring Boot detects the dependencies available on the classpath and configures Spring beans accordingly. There are a number of annotations, examples are @ConditionalOnClass, @ConditionalOnBean, @ConditionalOnMissingBean and @ConditionalOnMissingClass, that allows for applying conditions to Spring configuration classes or Spring bean declaration methods in such classes.

Examples:
- ✓ A Spring bean is to be created only if a certain dependency is available on the classpath. Use @ConditionalOnClass and supply a class contained in the dependency in question.
- ✓ A Spring bean is to be created only if there is no bean of a certain type or with a certain name created. Use @ConditionalOnMissingBean and specify name or type of bean to check.

What things affect what Spring Boot sets up?

There are a number of condition annotations in Spring Boot each of which can be used to control the creation of Spring beans. The following is a list of the condition annotations in Spring Boot (there are more):

Condition Annotation	Condition Factor
@ConditionalOnClass	Presence of class on classpath
@ConditionalOnMissingClass	Absence of class on classpath
@ConditionalOnBean	Presence of Spring bean or bean type (class)
@ConditionalOnMissingBean	Absence of Spring bean or bean type (class)
@ConditionalOnProperty	Presence of Spring environment property
@ConditionalOnResource	Presence of resource such as file
@ConditionalOnWebApplication	If the application is considered to be a web application, that is uses the Spring WebApplicationContext
@ConditionalOnNotWebApplication	If the application is not considered to be a web application

Creation of Spring Boot Application

```
package com.praveen.restservices;

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

@SpringBootApplication
public class UserManagementServiceApplication {

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

}
```

Spring Boot will do the following when you create the application

- ✓ Classpath Scan
- ✓ Default configuration setup
- ✓ Create an appropriate ApplicationContext instance
- ✓ Start embedded application server(for web application)

Let's see how Spring Boot internally works

Starter POM
META-INF/spring.factories
1.Enable
2.Disable

Based on @Conditional and @Configuration it will enable the respective component.

@SpringBootApplication

This is a combination of @SpringBootConfiguration,@EnableAutoConfiguration and @ComponentScan

@SpringBootConfiguration

Indicates that a class provides Spring Boot application @Configuration. Can be used as an alternative to the Spring's standard @Configuration annotation so that configuration can be found automatically

@EnableAutoConfiguration

This enables the component at runtime.
Spring Boot auto-configuration attempts to automatically configure your spring application based on their jar dependencies that you have added.

For example, in general in Spring MVC, DispatcherServlet and InternalResourceViewResolver will be configured in web.xml or we will go with Java Based configuration.
In Spring Boot, DispatcherServlet and InternalResourceViewResolver are autoconfigured to avoid boiler plate configuration.

@ComponentScan

This will be used by Spring IOC container to scan the packages for fetching the bean.

Why we need main method in Spring Boot application

Spring Boot Application executes as a standalone application so a main method is necessary and it helps to deploy the application jar in the embedded tomcat.
SpringApplication.run(…) internal flow
- ✓ Create application context
- ✓ Check the application type
- ✓ Register the annotated class beans with the context
- ✓ Creates an instance of TomcatEmbeddedServletContainer and adds the context

Spring Boot Starters

Starters are aggregate grouping of multiple dependencies into a single dependency.

In Spring Boot application, we configure Spring Boot Starter as a dependency in pom.xml. It will automatically add the spring jars and all other compatible dependencies to the class path of spring application.

Some of the spring boot starters are
- ✓ spring-boot-starter-parent
- ✓ spring-boot-starter-web

- ✓ spring-boot-starter-jdbc
- ✓ spring-boot-starter-test
- ✓ spring-boot-starter-security
- ✓ spring-boot-starter-data-jpa
- ✓ spring-boot-starter-activemq
- ✓ spring-boot-starter-amqp
- ✓ spring-boot-starter-data-redis
- ✓ spring-boot-starter-actuator

To create a Spring Boot Application using Spring Boot support then you have to make your application as a child of Spring Boot parent application

In pom.xml, we add the following

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.8.RELEASE</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>
```

In how many ways we can call the SpringBoot application?

- ✓ Using Spring CLI
- ✓ Using STS inbuilt tomcat
- ✓ By implementing the CommandLineRunner

Using CommandLineRunner

```java
@SpringBootApplication
public class PraveenorugantiSpringbootJpaHibernateApplication implements CommandLineRunner {
private Logger log = LoggerFactory.getLogger(this.getClass());

    @Autowired
    EmployeeRepository employeeRepository;

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

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

        employeeRepository.insert(new FullTimeEmployee("Praveen", new BigDecimal("100000")));
        employeeRepository.insert(new PartTimeEmployee("Naveen", new BigDecimal("50")));

        log.info("PartTime Employees {}"+employeeRepository.retrievePartTimeEmployees());
        log.info("FullTime Employees {}"+employeeRepository.retrieveFullTimeEmployees());


    }

}
```

Spring Boot Profile

You can define default configuration in application.properties. Environment specific overrides can be configured in specific files:
application-dev.properties
application-qa.properties
application-stage.properties
application-prod.properties

Setting A Spring Boot Profile
Here are a couple of ways of setting the active profile:

1. At the time of launching the Java application
-Dspring.profiles.active=qa – in the VM properties, OR

2. Do the following in the application.properties file
spring.application.profiles=qa.

Depending on which profile is currently the active, the appropriate configuration is picked up.

Using Profiles In Code

A profile can be used in code to define your beans. For example, have a look at the following piece of code:

```
@Profile("dev")
@Bean
public String devBean() {
return "I will be available in profile dev";
}

@Profile("prod")
@Bean
public String prodBean() {
return "I will be available in profile prod";
}
```

The bean devBean() will only be available with the dev profile, as it has been annotated with @Profile("dev"). Similarly, the bean prodBean() is only available with the profile prod.

In SpringBoot we can use properties file or YAML file for configurations but as per my view YAML is preferable

What is YAML?

YAML is a superset of JSON, and as such is a very convenient format for specifying hierarchical configuration data.

From YAML site : It is a human friendly data serialization standard for all programming languages.

YAML is more readable and it is good for the developers for read/write configuration files.

Here with the sample YAML file

```yaml
application-dev.yml
 1 ---
 2
 3 eureka:
 4   client:
 5     registerWithEureka: true
 6     fetchRegistry: true
 7     serviceUrl:
 8       defaultZone: http://localhost:8761/eureka/
 9   instance:
10     hostname: localhost
11
12 management:
13   endpoint:
14     refresh:
15       enabled: true
16     health:
17       show-details: always
18   endpoints:
19     web:
20       exposure:
21         include: "*"
22
23 server:
24   port: 8080
25   error:
26     include-stacktrace: never
27
```

Developer Tools

SpringBoot provides devtools for developer instead of restarting the server every time when we make a change in our dev environment.

We need to configure below dependency for devtools

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

Logging

Logging is a very important part of any application and it helps with debugging issues. Spring Boot, by default, includes spring-boot-starter-logging as a transitive dependency for the spring-boot-starter module. By default, Spring Boot includes SLF4J along with Logback implementations. Spring Boot has a LoggingSystem abstraction that automatically configures logging based on the logging configuration files

available in the classpath.

If Logback is available, Spring Boot will choose it as the logging handler. You can easily configure logging levels within the application.properties file without having to create logging provider specific configuration files such as logback.xml or log4j.properties.

```
logging:
    path: D:\
    file: user-management-service.log
    level:
      root: INFO
      org:
        springframework:
          security: INFO
          jdbc:
            core: TRACE
```

If you want to have more control over the logging configuration, create the logging provider specific configuration files in their default locations, which Spring Boot will automatically use.

For example, if you place the logback.xml file in the root classpath, Spring Boot will automatically use it to configure the logging system.

```
X logback.xml ⋈
 1  <configuration>
 2  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
 3  <encoder>
 4  <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
 5  </encoder>
 6  </appender>
 7  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
 8  <file>app.log</file>
 9  <encoder>
10  <pattern>%date %level [%thread] %logger{10} [%file:%line] %msg%n </pattern>
11  </encoder>
12  </appender>
13  <logger name="com.apress" level="DEBUG" additivity="false">
14  <appender-ref ref="STDOUT" />
15  <appender-ref ref="FILE" />
16  </logger>
17  <root level="INFO">
18  <appender-ref ref="STDOUT" />
19  <appender-ref ref="FILE" />
20  </root>
21  </configuration>
```

Type-Safe Configuration Properties

Spring provides the @Value annotation to bind any property value to a bean property.

```yaml
praveen-user-management-service:
        rabbitmq:
            queueName: praveenmq
            topicExchange: praveenexchange
        redis:
            host: localhost
            port: 6379
            password:
            jedis:
             pool:
               max-active: 8
               max-idle: 8
               max-wait: -1
               min-idle: 0
```

```java
package com.praveen.restservices.config;

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

@Configuration
@Getter
@Setter
@RefreshScope
public class UserProperties {

    @Value("${praveen-user-management-service.redis.host}")
    private String hostName;

    @Value("${praveen-user-management-service.redis.port}")
    private int port;

    @Value("${praveen-user-management-service.redis.password}")
    private String password;

    @Value("${praveen-user-management-service.redis.jedis.pool.max-active}")
    private int jedisPoolMaxActive;

    @Value("${praveen-user-management-service.redis.jedis.pool.max-idle}")
    private int jedisPoolMaxIdle;

    @Value("${praveen-user-management-service.redis.jedis.pool.max-wait}")
    private int jedisPoolMaxWait;

    @Value("${praveen-user-management-service.redis.jedis.pool.min-idle}")
    private int jedisPoolMinIdle;


}
```

SpringBoot Actuator

Actuator in general is used to monitor and manage your SpringBoot application by providing production-ready features like health check-up, auditing, metrics gathering, HTTP tracing etc. All of these features can be accessed over JMX or HTTP endpoints.

How to Enable SpringBoot Actuator?

It's very simple, just you need to add below dependencies in pom.xml

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

Open https://localhost:8080/actuator and if it asks creditionals then give username as praveen2 and password as pcf2

1. /health endpoint

Let's explore the health endpoint by opening the https://localhost:8080/actuator/health

The status will be UP as long as the application is healthy. It will show DOWN if the application gets unhealthy due to any issue like connectivity with the database or lack of disk space etc.

If we need to display the detailed health information then we need to configure below in application.yml

```yaml
management:
  endpoint:
    health:
      show-details: always
```

Let's open the health endpoint https://localhost:8080/actuator/health and see what comes up after inclusion of the above property.

```
{
    status: "UP",
    - details: {
        - clientConfigServer: {
            status: "UP",
            - details: {
                - propertySources: [
                    "configClient",
                    "https://github.com/praveenorugantl/praveen-spring-config-server/dev/application.yml"
                ]
            }
        },
        - diskSpace: {
            status: "UP",
            - details: {
                total: 1073741824,
                free: 886689792,
                threshold: 10485760
            }
        },
        - db: {
            status: "UP",
            - details: {
                database: "MySQL",
                hello: 1
            }
        },
        - refreshScope: {
            status: "UP"
        },
        - discoveryComposite: {
            status: "UP",
            - details: {
                - discoveryClient: {
                    status: "UP"
```

By default only health and info endpoints will be enabled over HTTP. If you want to enable all endpoints then you need to configure below in application.yml

```yaml
management:
    endpoints:
        web:
            exposure:
                include: "*"
```

```
{
    - _links: {
        - self: {
            href: "https://praveen-user-management-service.cfapps.io/actuator",
            templated: false
        },
        - archaius: {
            href: "https://praveen-user-management-service.cfapps.io/actuator/archaius",
            templated: false
        },
        - auditevents: {
            href: "https://praveen-user-management-service.cfapps.io/actuator/auditevents",
            templated: false
        },
        - beans: {
            href: "https://praveen-user-management-service.cfapps.io/actuator/beans",
            templated: false
        },
        - caches: {
            href: "https://praveen-user-management-service.cfapps.io/actuator/caches",
            templated: false
        },
        - caches-cache: {
            href: "https://praveen-user-management-service.cfapps.io/actuator/caches/{cache}",
            templated: true
        },
        - health-component: {
            href: "https://praveen-user-management-service.cfapps.io/actuator/health/{component}",
            templated: true
        },
```

2. /metrics endpoint

The /metrics endpoint lists all the metrics that are available for you to track.

https://localhost:8080/actuator/metrics/

```
{
  - names: [
      "jvm.threads.states",
      "http.server.requests",
      "process.files.max",
      "jvm.gc.memory.promoted",
      "jdbc.connections.active",
      "jvm.memory.committed",
      "system.load.average.1m",
      "http.client.requests",
      "jvm.memory.used",
      "jvm.gc.max.data.size",
      "jdbc.connections.max",
      "jdbc.connections.min",
      "system.cpu.count",
      "logback.events",
      "tomcat.global.sent",
      "jvm.buffer.memory.used",
      "tomcat.sessions.created",
      "jvm.memory.max",
      "jvm.threads.daemon",
      "system.cpu.usage",
      "jvm.gc.memory.allocated",
      "tomcat.global.request.max",
      "hikaricp.connections.idle",
      "hikaricp.connections.pending",
      "tomcat.global.request",
      "tomcat.sessions.expired",
      "hikaricp.connections",
      "jvm.threads.live",
      "jvm.threads.peak",
      "tomcat.global.received",
      "hikaricp.connections.active",
```

To get the details of an individual metric, you need to pass the metric name in the URL like this (https://localhost:8080/actuator/metrics/{MetricName})

For example, to get the details of system.cpu.usage metric, use the URL https://localhost:8080/actuator/metrics/system.cpu.usage. This will display the details in JSON format like so

```
{
    name: "system.cpu.usage",
    description: "The "recent cpu usage" for the whole system",
    baseUnit: null,
  - measurements: [
      - {
            statistic: "VALUE",
            value: 0.17336099075457978
        }
    ],
    availableTags: [ ]
}
```

3. /loggers endpoint

The loggers endpoint, which can be accessed at https://localhost:8080/actuator/loggers

```
{
  - levels: [
        "OFF",
        "ERROR",
        "WARN",
        "INFO",
        "DEBUG",
        "TRACE"
  ],
  - loggers: {
      - ROOT: {
            configuredLevel: "INFO",
            effectiveLevel: "INFO"
      },
      - com: {
            configuredLevel: null,
            effectiveLevel: "INFO"
      },
      - com.netflix: {
            configuredLevel: null,
            effectiveLevel: "INFO"
      },
      - com.netflix.appinfo: {
            configuredLevel: null,
            effectiveLevel: "INFO"
      },
      - com.netflix.appinfo.ApplicationInfoManager: {
            configuredLevel: null,
            effectiveLevel: "INFO"
      },
      - com.netflix.appinfo.InstanceInfo: {
            configuredLevel: null,
            effectiveLevel: "INFO"
      },
```

You can also view the details of an individual logger by passing the logger name in the URL like this

https://localhost:8080/actuator/loggers/root

```
{
    configuredLevel: "INFO",
    effectiveLevel: "INFO"
}
```

You can also change logger levels at runtime by making a POST request on URL

https://localhost:8080/actuator/loggers/root

4. /info endpoint

Now let's concentrate on /info endpoint. For this we need to configure the below in application.yml

Now open the below URL and you will see the required info as configured in yml file
https://localhost:8080/actuator/info

```
{
- app: {
    name: "SpringBoot Restful service",
    description: "SpringBoot Restful service",
    version: "1.0",
    encoding: "UTF-8",
  - java: {
      version: "1.8.0_211"
    }
  }
}
```

5. /beans endpoint

The /beans endpoint shows all the beans registered in your application, including the beans you explicitly configured and the beans autoconfigured by Spring Boot.
Now open the URL: http://localhost:8080/application

6. /autoconfig endpoint

The /autoconfig endpoint shows the autoconfiguration report, which is categorized into positiveMatches and negativeMatches.
Now open the URL: http://localhost:8080/application/autoconfig

7. /mappings endpoint

The /mappings endpoint shows all the @RequestMapping paths declared in the application. This is very helpful for checking which request path will be handled by which controller method.
Now open the URL http://localhost:8080/application/mappings

8. /configprops endpoint

The /configprops shows all the configuration properties defined by the @ConfigurationProperties beans,including your own configuration properties defined in the application.properties or YAML files.
Now open the URL http://localhost:8080/application/configprops

9. /env endpoint

The /env endpoint will expose all the properties from the Spring's ConfigurableEnvironment interface, such as a list of active profiles, application properties, system environment variables, and so on.
Now open the URL http://localhost:8080/application/env

10. /trace endpoint

The /trace endpoint shows the tracing information of the last few HTTP requests, which is very helpful for debugging the request/response details, like headers, cookies, etc.
Now open the URL http://localhost:8080/application/trace to view the HTTP request tracing details.

11. /dump endpoint

You can view the thread dump of your application with the details of the running threads and the stack trace of the JVM at http://localhost:8080/application/dump

12. /logfile endpoint

If you enabled file-based logging by setting logging.file or logging.path or using the native file configuration files (logback.xml, log4j.properties, etc.), you can use the /logfile endpoint to view the log file content. Go to http://localhost:8080/application/logfile

13. /shutdown endpoint

The /shutdown endpoint can be used to gracefully shut down the application, which is not enabled by default. You can enable this endpoint by adding the following property to application.properties

endpoints.shutdown.enabled=true

After adding this property, you can send the HTTP POST method to http://localhost:8080/application/shutdown to invoke the /shutdown endpoint.

Once the /shutdown endpoint is invoked successfully, you should see the following message:
{
"message": "Shutting down, bye..."
}

Note Be careful about enabling /shutting down an endpoint. Enable or shut down an endpoint only when it is absolutely required and be sure to protect the endpoint with the appropriate security configuration

14. /actuator endpoint

The /actuator endpoint provides a hypermedia-based "discovery page" for the other endpoints. To activatethis endpoint, you need to have the following Spring HATEOAS dependency.
<dependency>
<groupId>org.springframework.hateoas</groupId>
<artifactId>spring-hateoas</artifactId>
</dependency>
Go to http://localhost:8080/application/ to see the list of actuator endpoints

You can customize the actuator endpoint URL by setting the endpoints.actuator.path property.
endpoints.actuator.path=/actuator
Now you can access the Actuator endpoint at http://localhost:8080/application/actuator

Customizing Actuator Endpoints
By default, the Spring Boot Actuator endpoints run on the same port and the default management contextpath is "/application". You can customize these properties using the following properties.

management.context-path=/management
management.port=9090

Securing Actuator Endpoints
By default all sensitive endpoints are secured and only authenticated users who have the ACTUATOR role can access those endpoints. You can change the ACTUATOR role name to something else, say SUPERADMIN, by setting the following property.

management.security.roles=SUPERADMIN

If you have the Spring Boot Security starter on the classpath, the Actuator endpoints will be secured by Spring Security.

Add the Security starter dependency to pom.xml.
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId>
</dependency>

Instead of using the default user credentials, you can configure the security user credential in application.properties as follows.
security.user.name=admin
security.user.password=secret
security.user.role=USER,ADMIN,ACTUATOR
Now if you try to access any endpoint, say http://localhost:8080/application/beans, you will be prompted to enter credentials.
But most likely you will be using a custom Spring Security configuration backed by a datastore for user credentials, so you can configure Actuator endpoints for security as needed.
If, for any reason, you want to disable security for your Actuator endpoints, you can set the following property:

management.security.enabled=false

This will disable security for all Actuator endpoints. You are strongly advised to secure the Actuator endpoints, especially if your application is publicly accessible.

HAL browser

HAL browser is used to go through all HTTP endpoints used by our SpringBoot application internally

How Will You Monitor Multiple Microservices For Various Indicators Like Health?

Spring Boot provides actuator endpoints to monitor metrics of individual Microservices. These endpoints are very helpful for getting information about applications like if they are up, if their components like database etc are working good. But a major drawback or difficulty about using actuator endpoints is that we have to individually hit the endpoints for applications to know their status or health.

Imagine Microservices involving 50 applications, the admin will have to hit the actuator endpoints of all 50 applications. To help us deal with this situation, we will be using open source project located at Built on top of Spring Boot Actuator, it provides a web UI to enable us visualize the metrics of multiple applications.

Let's see how we can create spring boot admin server application.

Let's add below dependency in pom.xml

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
    <version>2.1.6</version>
</dependency>
```

Let's create Main class of spring boot admin server application

```
SpringbootAdminServerApplication.java ☒
 1  package com.praveen.springboot.admin;
 2
 3⊕ import org.springframework.boot.SpringApplication;
 8
 9  @EnableAdminServer
10  @SpringBootApplication
11  @EnableEurekaClient
12  public class SpringbootAdminServerApplication {
13
14⊖    public static void main(String[] args) {
15        SpringApplication.run(SpringbootAdminServerApplication.class, args);
16    }
17
18  }
```

We need to configure the eureka properties in application.yml

```yaml
application.yml

1 server:
2     port: 9111
3
4 spring:
5   application:
6     name: praveenoruganti-springboot-admin-server
7
8
9 eureka:
10   client:
11     registerWithEureka: true
12     fetchRegistry: true
13     serviceUrl:
14       defaultZone: http://localhost:8761/eureka/
15   instance:
16     hostname: localhost
17
18 management:
19   endpoints:
20     web:
21       exposure:
22         include: "*"
23   endpoint:
24     health:
25       show-details: ALWAYS
```

Spring Security

What are authentication and authorization? Which must come first?

The short explanation of authentication is that it is the process of verifying that, for instance, a user of a computer system is who he/she claims to be. In Spring Security, the authentication process consists of the following steps quoted from the Spring Security reference:

- ✓ The username and password are obtained and combined into an instance of UsernamePasswordAuthenticationToken (an instance of the Authentication interface).
- ✓ The token is passed to an instance of AuthenticationManager for validation.
- ✓ The AuthenticationManager returns a fully populated Authentication instance on successful authentication.
- ✓ The security context is established by calling SecurityContextHolder.getContext().setAuthentication(…), passing in the returned authentication object.

Authorization

Authorization is the process of determining that a user is permitted to do something that the user is attempting to do. Authorization is the process of specifying access rights to resources.

Authorization typically involves the following two separate aspects that combine to describe the accessibility of the secured system:

- ✓ The first is the mapping of an authenticated principal to one or more authorities (often called roles);
- ✓ The second is the assignment of authority checks to secured resources of the system.

Which must come first?

Unless there is some type of authorization that specifies what resources and/or functions that can be accessed by anonymous users, authentication must always come before authorization.

Application Security Framework

1. Login and Logout Functionality
2. Allow/block access to URLs to logged in users
3. Allow/block access to URLs to logged in users and with certain roles

Handles common vulnerabilities

1. Session fixation
2. Clickjacking
3. Click site forgery

With Spring Security we can do
1. Username/password Authentication
2. SSO/Okta/LDAP
3. App Level Authorization
4. Intra App Authorization like OAuth
5. Microservice security(using tokens,JWT)
6. Method Level Security

5 Core concepts in Spring Security

1. Authentication (Who is this User?)
Establishing that a principal's credentials are valid.

2. Authorization (Are they allowed to do this?)
Deciding if a principal is allowed to perform an action.
Authentication comes first before Authorization because authorization process needs principal object with authority votes to decide user allow to perform a action for secured resource.

3. Principal (currently logged in user(per request) and each user have multiple ID's)
The principal is the currently logged in user.

4. Granted Authority (fine grain permissions of what user can do)
GrantedAuthority as an individual privilege for example could include READ_AUTHORITY, WRITE_PRIVILEGE, or even CAN_EXECUTE_AS_ROOT.
When using a GrantedAuthority directly, such as through the use of an expression like hasAuthority('READ_AUTHORITY'), we are restricting access in a fine-grained manner.

5. Roles(group of authorities that can be assigned to a user i.e.. coarse grained permissions(i.e.. grouping of permissions)
We can think of each Role as a coarse-grained GrantedAuthority that is represented as a String and prefixed with "ROLE". When using a Role directly, such as through an expression like hasRole("ADMIN"), we are restricting access in a coarse-grained manner.

How to add spring security to a spring boot application?

1. Add the below dependency in pom.xml

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

2. Once i add the above spring security dependency, the security will be enabled by default.
Spring generally uses Filters for providing the default security mechanism.

Spring Security default behaviour
- ✓ Adds mandatory authentication for all URL's
- ✓ Adds login form
- ✓ Handles login error
- ✓ Creates a user and sets a default password

Once you hit URL: http://localhost:8080/, u will see below login form

Now whats the user id and password i need to provide?

As you have not configured user id and password, at starup of sping boot application, spring security component by default generates a **password** and username is **user** as shown in below screenshot.

Now lets provide the username as **user** and password as **04547a8d-b3a5-4de9-8261-6e74e8752370** and click on Sign in button.

It looks like below screenshot and we are able to login into the application without any issues.

Now lets set our customized user and password in application.yml

```
spring:
    application:
      name: praveen-user-management-service
    profiles:
     active: dev
    cloud:
     config:
      uri: http://localhost:8888
    security:
     user:
        name: praveen
        password: pcf
```

Now restart the application and you will not be seeing the generated password from spring security component and i will be able to login into application using my customized user as **praveen** and password as **pcf**

How to configure authentication in spring security?

Using In-memory Authentication configuration

Steps:
1. Get Hold of AuthenticationManagerBuilder
2. Set the configuration on it.
How to get hold of AuthenticationManagerBuilder ??

There is a class called WebSecurityConfigurerAdapter.java which we need to extend and override the configure(AuthenticationManagerBuilder auth).

Now I will remove the default user id and password which i provided in my earlier step so that I can multiple user, password and role for the same.
As this is a sample, i am using NoOpPasswordEncoder for encoding the password.

```
@Configuration
@EnableWebSecurity
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter{

    // Authentication based on role
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("praveen")
                .password("pcfadmin")
                .roles("ADMIN")
                .and()
                .withUser("prasad")
                .password("pcfuser")
                .roles("USER");
    }

    @Bean
    public PasswordEncoder getPasswordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    }
}
```

How to configure authorization in spring security?

For this we need to get hold of HttpSecurity and this can be done by extending
the WebSecurityConfigurerAdapter.java and override the configure(HttpSecurity http)
method.

```
@Configuration
@EnableWebSecurity
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter{

    // Authentication based on role
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("praveen")
                .password("pcfadmin")
                .roles("ADMIN")
                .and()
                .withUser("prasad")
                .password("pcfuser")
                .roles("USER");
    }

    @Bean
    public PasswordEncoder getPasswordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    }

    // Authorization based on role
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/rest/hello/getMsg/admin").hasRole("ADMIN")
                .antMatchers("/rest/hello/getMsg/user").hasAnyRole("USER","ADMIN")
                .antMatchers("/").permitAll()
                .and().formLogin();
    }
}
```

**Please remember if you want to logout from the session then
use http://localhost:8080/logout**

How does spring authentication works internally?

boostrap authentication

This is done using filters which does the magic here.

The o.s.web.filter.DelegatingFilterProxy class implements the javax.servlet.Filter interface and thus is a servlet filter. DelegatingFilterProxy is a special servlet filter that, by itself, doesn't do much. Instead, it delegates all work to a Spring bean from the ApplicationContext root, which must implement javax.servlet.Filter. Since by default the bean is looked up by name, using the <filter-name> value, we must ensure we use springSecurityFilterChain as the value of <filter-name>. The pseudocode for how o.s.web.filter.DelegatingFilterProxy works for our web.xml file can be found in the following code snippet:

```xml
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

```java
public class DelegatingFilterProxy implements Filter {
  void doFilter(request, response, filterChain) {
    Filter delegate = applicationContet.getBean("springSecurityFilterChain")
    delegate.doFilter(request,response,filterChain);
  }
}
```

What is the security filter chain?

The security filter chain implements the SecurityFilterChain interface and the only implementation provided by Spring Security is the DefaultSecurityFilterChain class. The constructor of the DefaultSecurityFilterChain class takes a variable number of parameters, the first always being a request matcher. The remaining parameters are all filters which implements the javax.servlet.Filter interface.

What is a security context?

The most fundamental object is SecurityContextHolder. This is where we store details of the present security context of the application, which includes details of the principal currently using the application. By default the SecurityContextHolder uses a ThreadLocal to store these details, which means that the security context is always

available to methods in the same thread of execution, even if the security context is not explicitly passed around as an argument to those methods.

SecurityContextHolder.getContext().

The object returned by the call to getContext() is an instance of the SecurityContext interface.

Taking a look at the SecurityContext interface, which defines the minimum security information associated with a thread of execution, there are two methods; one for setting and one for retrieving an object that implements the Authentication interface.

Authentication

In general for Authentication the inputs will be creditionals and output will be principle. AuthenticationProvider authenticate(Authentication authentication) method does the above.

The Authentication interface defines the properties of an object that represents a security token for:

- ✓ A collection of the authorities granted to the principal
- ✓ The credentials used to authenticate a user. This can be a login name and a password that has been verified to match
- ✓ Details Additional information, may be application specific or null if not used.
- ✓ Principal
- ✓ Authenticated flag A boolean indicating whether the principal has been successfully authenticated

UserDetails is not used for security purposes, it is just a "user info" bean. Spring Security uses Authentication instances. So Authentication instance will usually have only the information needed to let users log in (usernames, credentials and roles, basically). UserDetails is more generic, and can include anything related to user management (such as contact information, account information, photographs, whatever).

Typically, you will have an Authentication instance backed by a UserDetails instance.

org.springframework.security.authentication

Interface AuthenticationProvider

```
public interface AuthenticationProvider
```

Indicates a class can process a specific Authentication implementation.

Method Summary

| All Methods	Instance Methods	Abstract Methods
Modifier and Type	**Method and Description**	
Authentication	authenticate(Authentication authentication) Performs authentication with the same contract as AuthenticationManager.authenticate(Authentication).	

org.springframework.security.core

Interface Authentication

Method Summary

All Methods	Instance Methods	Abstract Methods	
Modifier and Type		Method and Description	
java.util.Collection<? extends GrantedAuthority>		getAuthorities() Set by an AuthenticationManager to indicate the authorities that the principal has been granted.	
java.lang.Object		getCredentials() The credentials that prove the principal is correct.	
java.lang.Object		getDetails() Stores additional details about the authentication request.	
java.lang.Object		getPrincipal() The identity of the principal being authenticated.	
boolean		isAuthenticated() Used to indicate to AbstractSecurityInterceptor whether it should present the authentication token to the AuthenticationManager.	
void		setAuthenticated(boolean isAuthenticated) See isAuthenticated() for a full description.	

org.springframework.security.core.userdetails

Interface UserDetails

Method Summary

All Methods	Instance Methods	Abstract Methods	
Modifier and Type		Method and Description	
java.util.Collection<? extends GrantedAuthority>		getAuthorities() Returns the authorities granted to the user.	
java.lang.String		getPassword() Returns the password used to authenticate the user.	
java.lang.String		getUsername() Returns the username used to authenticate the user.	
boolean		isAccountNonExpired() Indicates whether the user's account has expired.	
boolean		isAccountNonLocked() Indicates whether the user is locked or unlocked.	
boolean		isCredentialsNonExpired() Indicates whether the user's credentials (password) has expired.	
boolean		isEnabled() Indicates whether the user is enabled or disabled.	
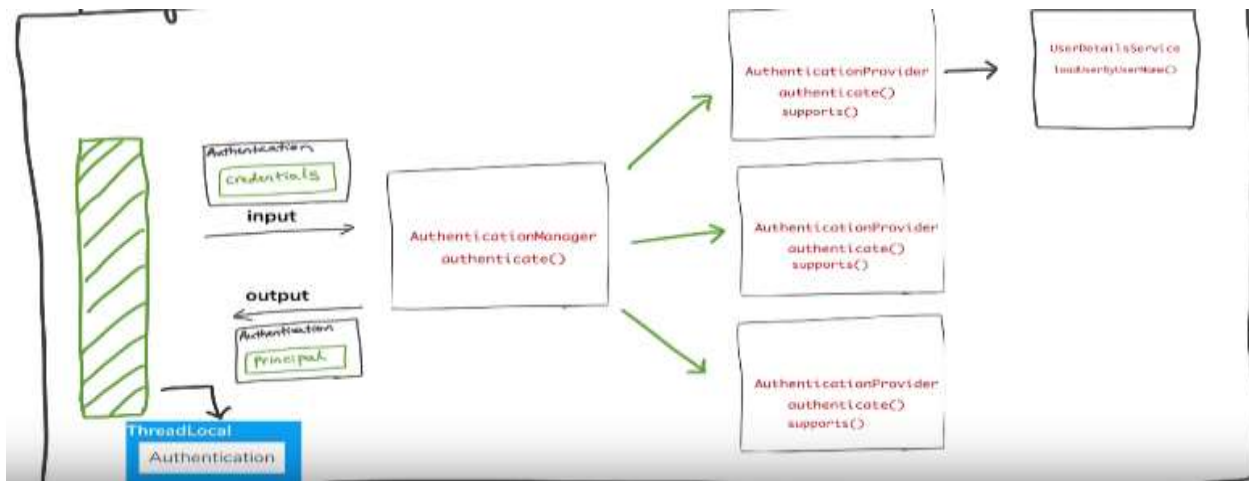
How to setup JDBC Authentication?

This is very simple you first need to configure the datasource and set appropriate properties in application.yml

```
datasource:
        type: com.zaxxer.hikari.HikariDataSource
        driver-class-name: com.mysql.cj.jdbc.Driver
        url: jdbc:mysql://root:password@localhost:3306/praveendb?reconnect=true
        username: root
        password: password
        hikari:
            connectionTimeout : 30000
            idleTimeout  : 600000
            maxLifetime  : 1800000
            maximumPoolSize : 5
```

Then make sure to create the **user schema in the database.**

create table users(
    username varchar(50) not null primary key,
    password varchar(50) not null,
    enabled boolean not null
);
create table authorities (
    username varchar(50) not null,
    authority varchar(50)not null,
    constraint fk_authorities_users foreign key(username) references users(username)
);
create unique index ix_auth_username on authorities (username,authority);

Ref URL: https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#user-schema

Then go ahead and override the configure(AuthenticationManagerBuilder auth) by extending the WebSecurityConfigurerAdapter.java.
Then make sure you provide @EnableWebSecurity annotation on top of implementation class.

```
@Configuration
@EnableWebSecurity
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter{

    @Autowired
    DataSource datasource;

    // Authentication based on role
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        auth.jdbcAuthentication()
            .dataSource(datasource);
    }

    @Bean
    public PasswordEncoder getPasswordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    }

    // Authorization based on role
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/rest/hello/getMsg/admin").hasRole("ADMIN")
            .antMatchers("/rest/hello/getMsg/user").hasAnyRole("USER","ADMIN")
            .antMatchers("/").permitAll()
            .and().formLogin();
    }

}
```

Swagger2 implementation for a SpringBoot REST API

When creating a REST API, good documentation is instrumental.

Moreover, every change in the API should be simultaneously described in the reference documentation. Accomplishing this manually is a tedious exercise, so automation of the process was inevitable.

Adding below Maven dependencies in pom.xml for swagger2 and swagger-ui which makes user interaction with the Swagger-generated API documentation much easier.

```xml
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.7.0</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.7.0</version>
</dependency>
```

You need to include @EnableSwagger2 on top of main class and implement 2 methods i.e... configDock() and apiInfo().

```java
@SpringBootApplication
@EnableSwagger2
public class UserManagementServiceApplication {

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

    @Bean
    public Docket configDock() {
        return new Docket(DocumentationType.SWAGGER_2).select().apis(basePackage("com.praveen.restservices.controller"))
            .paths(regex("/rest.*")).build().apiInfo(apiInfo());
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder().title("PRAVEEN ORUGANTI SPRING BOOT SWAGGER")
            .description("WELCOME TO SWAGGER CLIENT")
            .contact(
                new Contact("PRAVEEN ORUGANTI", "https://praveenoruganti.blogspot.com/", "praveenoruganti@gmail.com"))
            .license("Apache 2.0").licenseUrl("http://www.apache.org/licenses/LICENSE-2.0.html").version("1.0.0")
            .build();
    }
}
```

Swagger2 Core Annotations

|	
@Api	Marks a class as a Swagger resource.
@ApiModel	Provides additional information about Swagger models.
@ApiModelProperty	Adds and manipulates data of a model property.
@ApiOperation	Describes an operation or typically an HTTP method against a specific path.
@ApiParam	Adds additional meta-data for operation parameters.
@ApiResponse	Describes a possible response of an operation.
@ApiResponses	A wrapper to allow a list of multiple ApiResponse objects.

Now configure above annotations in controller wherever applicable

```
@Api(tags = "User Management RESTful Services", value = "UserController", description = "Controller for User Management Service")
@RestController
@Validated
@RequestMapping(value = "/rest/users")
public class UserController {

    // Autowire the UserService
    @Autowired
    private UserService userService;

    // getAllUsers Method
    @ApiOperation(value = "Retrieve list of users")
    @GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
    public List<User> getUsers(@RequestParam(value="page", defaultValue = "1") int page,
                               @RequestParam(value="limit", defaultValue = "1") int limit,
                               @RequestParam(value="sort", defaultValue = "desc") String sort) {
        return userService.getAllUsers();
    }
}
```

Now you can test by providing URL: http://localhost:8080/swagger-ui.html
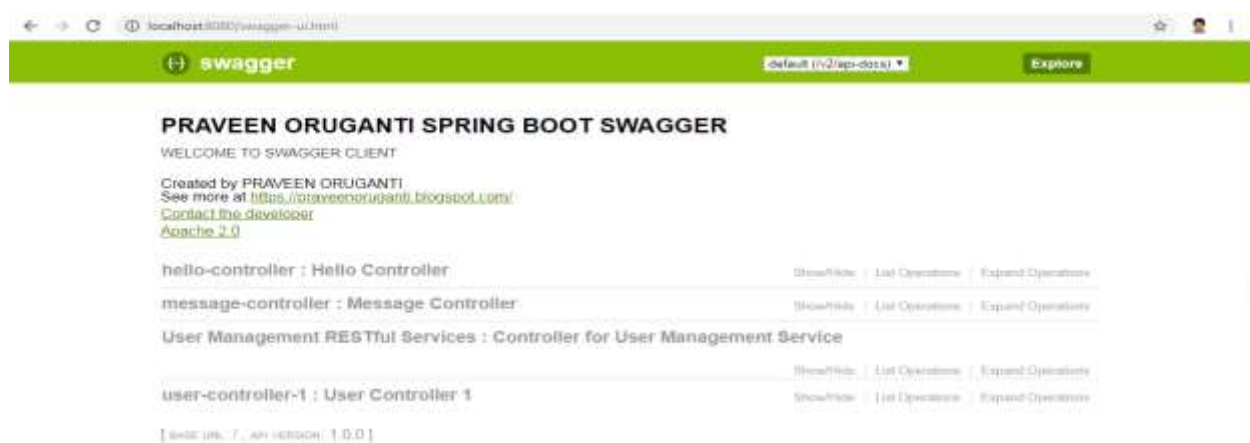
Let's see how Exception Handling done in Spring Boot Applications

Throwing exceptions is a great way to break your program flow when the required conditions are not met. For example, they could be used for validating input parameters.

When an unhandled exception occurs while processing a request, your API should handle it and send a suitable error response to the client with proper HTTP code and error details. Spring MVC provides multiple ways to do it. Among those, writing a controller advice looks like a good way, which allows us to code the handlers globally at a single place.

Solution 1

Before Spring 3.2, the main approach to handling exceptions in a Spring MVC application was @ExceptionHandler annotation.

What is @ExceptionHandler

@ExceptionHandler(value = Exception.class): Any method with this annotation will be called when an exception of type or the sub type of the class specified(Exception.class) is thrown in the Controllers.

```
@Controller
public class WebController {
        @ExceptionHandler(StudentNotFoundException.class)
 public ModelAndView handleStudentNotFoundException(StudentNotFoundException ex) {
  Map<String, String> model = new HashMap<String, String>();
  model.put("exception", ex.toString());
  return new ModelAndView("student.error", model);


 }
}
```

This approach has a major drawback – the @ExceptionHandler annotated method is only active for that particular Controller, not globally for the entire application. Of course, adding this to every controller makes it not well suited for a general exception handling mechanism.

Solution 2

The HandlerExceptionResolver

It will also allow us to implement a uniform exception handling mechanism in our REST API.

ExceptionHandlerExceptionResolver

This resolver was introduced in Spring 3.1 and is enabled by default in the DispatcherServlet. This is actually the core component of how the @ExceptionHandler mechanism presented earlier works.

DefaultHandlerExceptionResolve

This resolver was introduced in Spring 3.0 and is enabled by default in the DispatcherServlet. It is used to resolve standard Spring exceptions to their corresponding HTTP Status Codes.

ResponseStatusExceptionResolver

This resolver was also introduced in Spring 3.0 and is enabled by default in the DispatcherServlet. It's main responsibility is to use the @ResponseStatus annotation available on custom exceptions and to map these exceptions to HTTP status codes.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class MyResourceNotFoundException extends RuntimeException {
    public MyResourceNotFoundException() {
        super();
    }
    public MyResourceNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    public MyResourceNotFoundException(String message) {
        super(message);
    }
    public MyResourceNotFoundException(Throwable cause) {
        super(cause);
    }
}
```

Same as the DefaultHandlerExceptionResolver, this resolver is limited in the way it deals with the body of the response – it does map the Status Code on the response, but the body is still null.

Custom HandlerExceptionResolver

The combination of DefaultHandlerExceptionResolver and ResponseStatusExceptionResolver goes a long way towards providing a good error handling mechanism for a Spring RESTful Service. The downside is – as mentioned before – no control over the body of the response.

Ideally, we'd like to be able to output either JSON or XML, depending on what format the client has asked for via the Accept header.

The other important implementation detail is that we return a ModelAndView – this is the body of the response and it will allow us to set whatever is necessary on it.

```
@Component
public class RestResponseStatusExceptionResolver extends AbstractHandlerExceptionReso

    @Override
    protected ModelAndView doResolveException(
      HttpServletRequest request,
      HttpServletResponse response,
      Object handler,
      Exception ex) {
        try {
            if (ex instanceof IllegalArgumentException) {
                return handleIllegalArgument((IllegalArgumentException) ex, response,
            }
            ...
        } catch (Exception handlerException) {
            logger.warn("Handling of [" + ex.getClass().getName() + "]
              resulted in Exception", handlerException);
        }
        return null;
    }
```

Solution 3

After 3.2 we now have the new @ControllerAdvice annotation to address the limitations of the previous one.

What is @ControllerAdvice

Controller advices are classes that can contain global exception handler methods.

Controller advices can also contain additional things like global ModelAttributes.

A controller advice allows you to use similar exception handling techniques to @ExceptionHandler but apply them across the whole application, not just to an individual controller. You can think of them as an annotation driven interceptor.

Any class annotated with @ControllerAdvice becomes a controller-advice, then we have to write method that will handle specific exception:

```
@ControllerAdvice
public class RestResponseEntityExceptionHandler extends
  ResponseEntityExceptionHandler {
 @ExceptionHandler(value = { IllegalArgumentException.class, IllegalStateException.class })
    protected ResponseEntity<Object> handleConflict(RuntimeException ex, WebRequest request)
        String bodyOfResponse = "This should be application specific";
        return handleExceptionInternal(ex, bodyOfResponse,
          new HttpHeaders(), HttpStatus.CONFLICT, request);
    }
}
```

The new annotation allows the multiple scattered @ExceptionHandler from before to be consolidated into a single, global error handling component.

The actual mechanism is extremely simple but also very flexible:

- ✓ it allows full control over the body of the response as well as the status code
- ✓ it allows mapping of several exceptions to the same method, to be handled together
- ✓ it makes good use of the newer RESTful ResposeEntity response

One thing to keep in mind here is to match the exceptions declared with @ExceptionHandler with the exception used as the argument of the method. If these don't match, the compiler will not complain – no reason it should, and Spring will not complain either.

Solution 4

Spring 5 introduced the ResponseStatusException class. We can create an instance of it providing an HttpStatus and optionally a reason and a cause:

```java
// getUserById
@GetMapping("/{id}")
public User getUser(@PathVariable("id") @Min(1) Long id) {

    try {
        Optional<User> userOptional = userService.getUserById(id);
        return userOptional.get();
    } catch (UserNotFoundException ex) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, ex.getMessage());
    }

}
```

What are the benefits of using ResponseStatusException?

- ✓ Excellent for prototyping: We can implement a basic solution quite fast
- ✓ One type, multiple status codes: One exception type can lead to multiple different responses. This reduces tight coupling compared to the @ExceptionHandler
- ✓ We won't have to create as many custom exception classes
- ✓ More control over exception handling since the exceptions can be created programmatically

And what about the tradeoffs?

- ✓ There's no unified way of exception handling: It's more difficult to enforce some application-wide conventions, as opposed to @ControllerAdvice which provides a global approach
- ✓ Code duplication: We may find ourselves replicating code in multiple controllers

We should also note that it's possible to combine different approaches within one application.

For example, we can implement a @ControllerAdvice globally, but also ResponseStatusExceptions locally. However, we need to be careful: If the same exception can be handled in multiple ways, we may notice some surprising behavior. A possible convention is to handle one specific kind of exception always in one way.

```java
// Create User Method
// @RequestBody Annotation
// @PostMapping Annotation
@ApiOperation(value = "Creates a new user")
@PostMapping
public ResponseEntity<Void> createUser(
        @ApiParam("User information for a new user to be created.") @Valid @RequestBody User user,
        UriComponentsBuilder builder) {
    try {
        userService.createUser(user);
        HttpHeaders headers = new HttpHeaders();
        headers.setLocation(builder.path("/rest/users/{id}").buildAndExpand(user.getUserid()).toUri());
        return new ResponseEntity<Void>(headers, HttpStatus.CREATED);

    } catch (UserExistsException ex) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, ex.getMessage());
    }
}
```

```
@ResponseBody
@ExceptionHandler(HttpMediaTypeNotAcceptableException.class)
public String handleHttpMediaTypeNotAcceptableException() {
    return "acceptable MIME type:" + MediaType.APPLICATION_JSON_VALUE;
}
```

Spring Boot Testing

Below dependency is responsible for enabling the tests in springboot application.

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

Junit Testing

It's the most common practice for all testing related code to go in the src/test/java folder.

assertEquals	Checks if two primitive types or objects are equal.
assertTrue	Checks if input condition is true.
assertFalse	Checks if input condition is false.
assertNotNull	Checks if an object isn't null.
assertNull	Checks if an object is null.
assertSame	Checks if two object references point to the same object in memory.
assertNotSame	Checks if two object references do not point to the same object in memory.
assertArrayEquals	Checks whether two arrays are equal to each other.

```
package com.praveen.restservices.controller;

import static org.junit.Assert.assertEquals;

@RunWith(SpringJUnit4ClassRunner.class)
@ActiveProfiles("dev")
@SpringBootTest
public class HelloControllerTest {

    @Autowired
    private HelloController helloController;

    @Test
    public void testGreetingBean() {
        User1 user1 = helloController.greetingBean();
        assertEquals(user1.getUserEmail(), "praveenoruganti@gmail.com");
        assertEquals(user1.getAddress(), "Hyderabad");
        assertEquals(user1.getUserId(), Integer.valueOf(149903));
        assertEquals(user1.getUserName(), "PraveenOruganti");
    }

    @Test
    public void testGreeting() {
        String result = helloController.greeting();
        assertEquals(result, "praveen-user-management-service Welcome To Praveen Oruganti Forum !!");

    }
}
```

Let's use MockMVC

MockMVC class is part of Spring MVC test framework which helps in testing the controllers explicitly starting a Servlet container.

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ActiveProfiles("dev")
@SpringBootTest
public class HelloControllerTest {

    MockMvc mockMvc;

    @Autowired
    private HelloController helloController;

    @Before
    public void setup() throws Exception {
        this.mockMvc = standaloneSetup(this.helloController).build();
    }

    @Test
    public void testGreetingUser() throws Exception {
    this.mockMvc.perform(get("/rest/hello/getMsg/user"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$", is("praveen-user-management-service Welcome User To Praveen Oruganti Forum !!")));
    }

    @Test
    public void testGreetingMsgBean() throws Exception {
    this.mockMvc.perform(get("/rest/hello/getMsgBean/path-variable/Prasad"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$", is("<User1><userId>149903</userId><userName>Prasad</userName>"
                    + "<userEmail>praveenoruganti@gmail.com</userEmail><address>Hyderabad</address></User1>")));
    }
}
```

Please note Spring Rest, Spring Cloud, Microservices and PCF concepts will be covered in my other ebook.

Please check out my other ebooks in https://github.com/praveennaga/PraveenNaga-Tech-Ebooks

Buy me a Cup of Coffee:
https://docs.google.com/forms/d/e/1FAIpQLSfZxtgAxzMv83uwdEXezVFfLNSKlQgu7eDcMOeGtS2cRnOBrA/viewform?vc=0&c=0&w=1