Spring Boot

— Nandakumar Purohit

Spring Boot

Agenda

- Getting Started
- Building Hello World Application
- Spring Initializr
- Understanding Embedded Servers
- Using Spring Boot Actuator
- Building REST APIs using Spring Boot
- Designing your first REST API
- Creating a ToDo REST API
- Implementing Exception Handling for REST APIs
- Documenting REST Services using OpenAPI Spec

Spring Boot

Getting Started

- We created a web application with Spring MVC.
- Decide on the frameworks to use
- Configure the frameworks and integrate them.
- Exception handling,
- Application configuration and so on.

"But how about having all these for free when we create a new application?"

Spring Boot

Getting Started

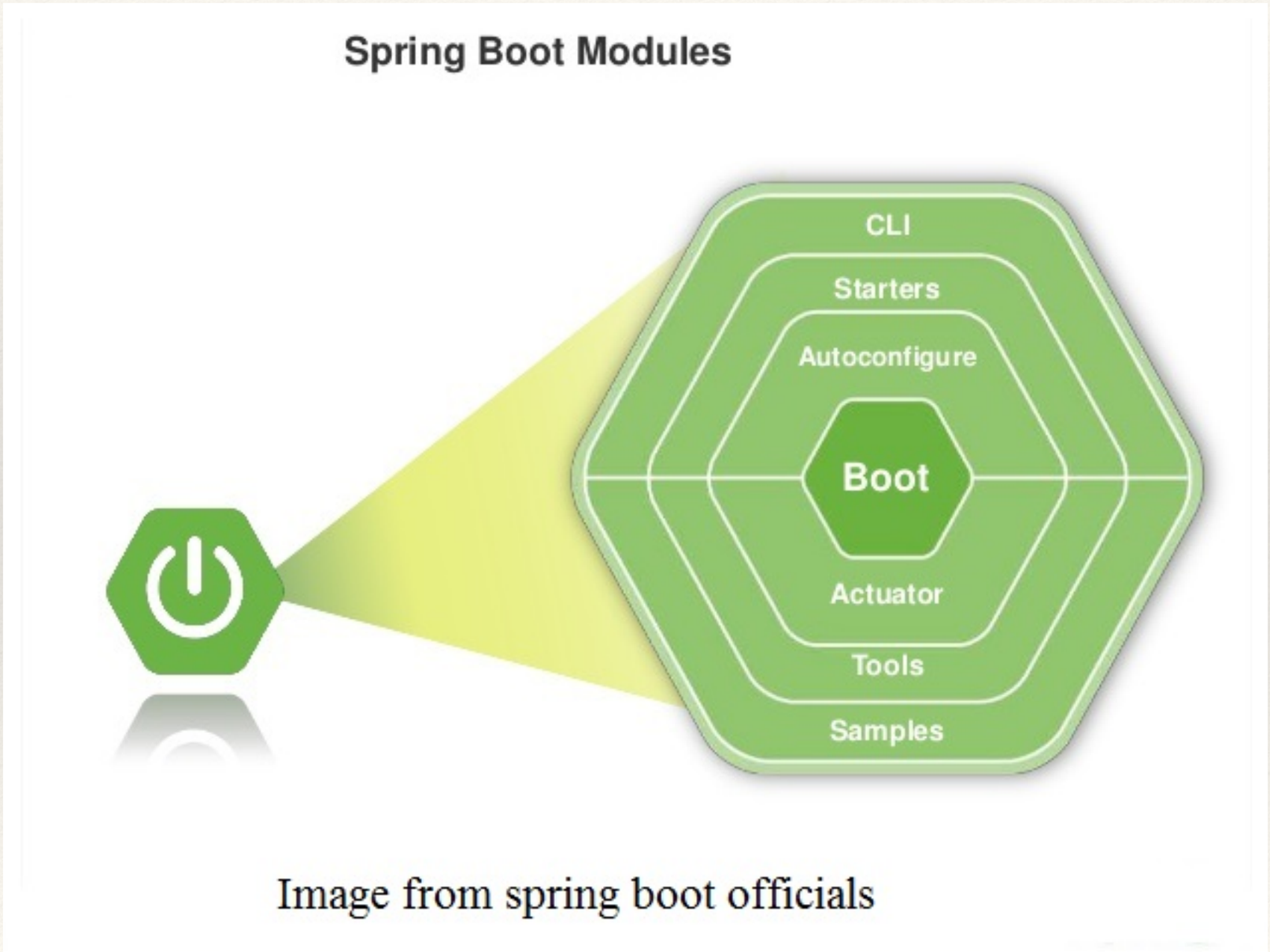


Image from spring boot officials

Spring Boot

Getting Started

Steps involved in building a prototype of an application - Spring MVC + Hibernate

1. Decide which versions of Spring MVC, JPA, and Hibernate to use.
2. Set up a Spring context to wire all the different layers together.
3. Set up a web layer with Spring MVC (including Spring MVC configuration):
    - Configure beans for `DispatcherServlet`, `handler`, `resolvers`, view resolvers, and so on
4. Set up Hibernate in the data layer:
    - Configure beans for `SessionFactory`, data source, and so on
5. Decide and implement how to store your application configuration, which varies between different environments.
6. Decide how you would want to perform your unit testing.
7. Decide and implement your transaction management strategy.
8. Decide and implement how to implement security.
9. Set up your logging framework.
10. Decide and implement how you want to monitor your application in production.
11. Decide and implement a metrics management system to provide statistics about the application.
12. Decide and implement how to deploy your application to a web or application server.

Spring Boot

Getting Started

Primary Goals of Spring Boot

- To allow you to **get off the ground quickly** with Spring-based projects.
- Be opinionated. **Make default assumptions** based on common usage.
- **Provide configuration options** to handle deviations from defaults.
- To provide a wide range of **non-functional features** out of the box.
- To **avoid using code generation** and **avoid using a lot of XML** configuration.

Non-functional Features

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

Spring Boot

Building Hello World Application

The following steps are involved in starting up with a Spring Boot application:

1. Configure `spring-boot-starter-parent` in your `pom.xml` file.
2. Configure the `pom.xml` file with the required starter projects.
3. Configure `spring-boot-maven-plugin` to be able to run the application.
4. Create your first Spring Boot `launch` class.

why do we need `spring-boot-starter-parent`?

- Contains the default versions of Java to use
- The default versions of dependencies that Spring Boot uses
- The default configuration of the Maven plugins

Spring Boot

Building Hello World Application

Understanding SpringApplication Class

The `SpringApplication` class can be used to Bootstrap and launch a Spring application from a Java `main` method.

1. Create an instance of Spring's **`ApplicationContext`**.
2. Enable the functionality to accept command-line arguments and expose them as Spring properties.
3. Load all the Spring beans as per the configuration.

The **`@SpringBootApplication`** annotation is a shortcut for three annotations:

- **`@Configuration`**: This indicates that this is a Spring application context configuration file.
- **`@EnableAutoConfiguration`**: This enables auto configuration, an important feature of Spring Boot. We will discuss auto configuration later in a separate section.
- **`@ComponentScan`**: This enables scanning for Spring beans in the package of this class and all its subpackages.

Spring Boot

Building Hello World Application

Understanding SpringApplication Class

- The Tomcat server is launched on port `8080`—`Tomcat started on port(s): 8080 (http)`.
- `DispatcherServlet` is configured. This means that the Spring MVC Framework is ready to accept requests – `Mapping servlet: 'dispatcherServlet' to [/]`.
- Four filters—`characterEncodingFilter`, `hiddenHttpMethodFilter`, `httpPutFormContentFilter`, and `requestContextFilter`—are enabled by default.

Problems @ Javadoc Declaration Console

demo-boot - DemoBootApplication [Spring Boot App] /Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/bin/java (Oct 19, 2019, 10:33:49 AM)

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.2.0.RELEASE)

2019-10-19 10:33:50.023  INFO 18654 --- [           main] com.example.demo.DemoBootApplication     : Starting DemoBootApplication on Nandakumars-MBP.home with PID 18654 (/Use
2019-10-19 10:33:50.025  INFO 18654 --- [           main] com.example.demo.DemoBootApplication     : No active profile set, falling back to default profiles: default
2019-10-19 10:33:50.530  INFO 18654 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2019-10-19 10:33:50.535  INFO 18654 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2019-10-19 10:33:50.535  INFO 18654 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.27]
2019-10-19 10:33:50.575  INFO 18654 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2019-10-19 10:33:50.575  INFO 18654 --- [           main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 519 ms
2019-10-19 10:33:50.671  INFO 18654 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2019-10-19 10:33:50.775  INFO 18654 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2019-10-19 10:33:50.777  INFO 18654 --- [           main] com.example.demo.DemoBootApplication     : Started DemoBootApplication in 0.948 seconds (JVM running for 1.298)
Hello!
```

Spring Boot

Building Hello World Application

Understanding the magic of AutoConfiguration

- **`basicErrorController`**, and **`handlerExceptionResolver`**: Basic exception handling. This shows a default error page when an exception occurs.
- **`beanNameHandlerMapping`**: Used to resolve paths to a handler (controller).
- **`characterEncodingFilter`**: Provides default character encoding UTF-8.
- **`dispatcherServlet`**: DispatcherServlet is the Front Controller in Spring MVC applications.
- **`jacksonObjectMapper`**: Translates objects to JSON and JSON to objects in REST services.
- **`messageConverters`**: The default message converters to convert from objects into XML or JSON, and vice versa.
- **`multipartResolver`**: Provides support to upload files in web applications.
- **`mvcValidator`**: Supports the validation of HTTP requests.
- **`viewResolver`**: Resolves a logical view name to a physical view.
- **`propertySourcesPlaceholderConfigurer`**: Supports the externalization of application configuration.
- **`requestContextFilter`**: Defaults the filter for requests.
- **`restTemplateBuilder`**: Used to make calls to REST services.
- **`tomcatEmbeddedServletContainerFactory`**: Tomcat is the default embedded servlet container for Spring Boot-based web applications.

Exploring Spring Boot Starter Projects

`spring-boot-starter-web-services`	This is a starter project to develop XML-based 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-starter-integration`	This supports the Spring Integration Framework, which 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-starter-validation`	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.

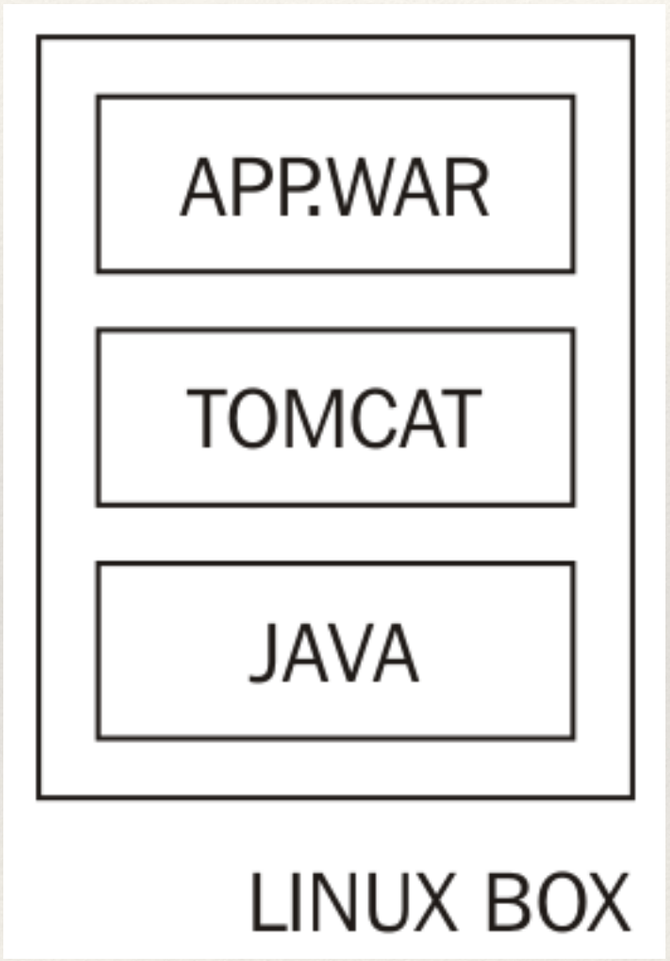
Exploring Spring Boot Starter Projects

`spring-boot-starter-websocket`	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-data-rest`	This is the support to expose REST services using Spring Data REST.

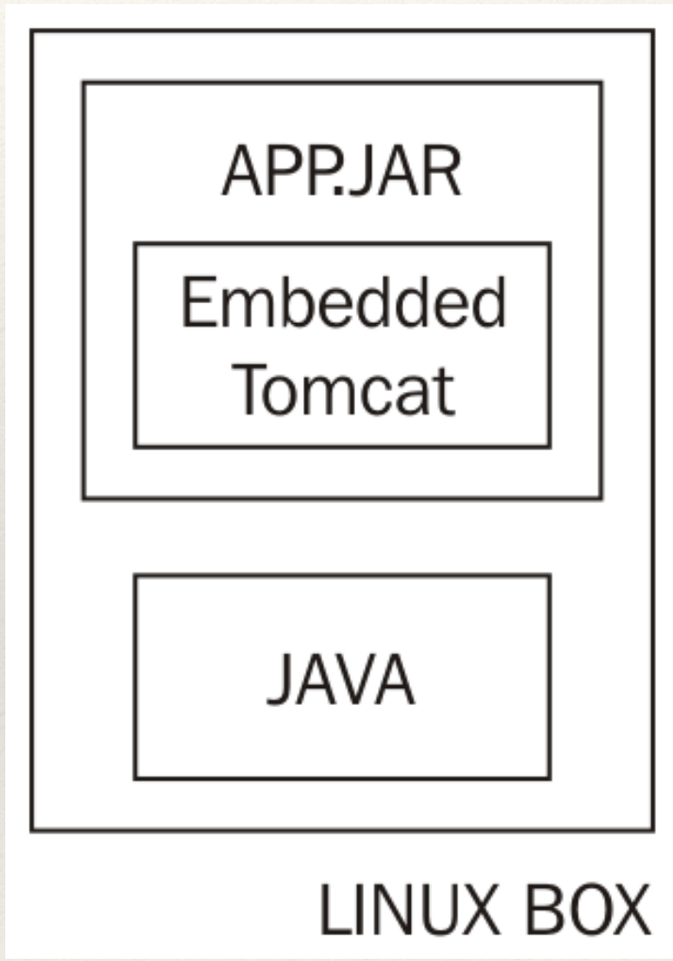
Spring Boot

Understanding Embedded Servers

Traditional Java App Deployment



Embedded Server Deployment



Spring Boot

Building Hello World Application

Switching to other embedded servers

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

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

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

JAR to WAR

```
<packaging>
war
</packaging>
```

Spring Boot

Using Developer Tools

We need to include a dependency:

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

- By default, disables the caching of view templates and static files.
- This allows 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.

- **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**

The advantages of Spring Boot developer tools are as follows:

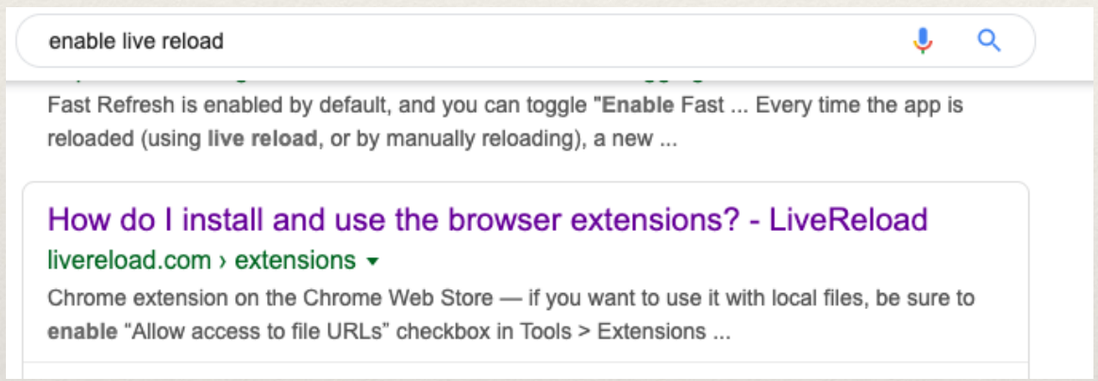
- No need to stop and start the application each time.
- The application is **automatically restarted** as soon as there is a change.
- It only reloads the actively developed classes. It does not reload the third-party JARs

Spring Boot

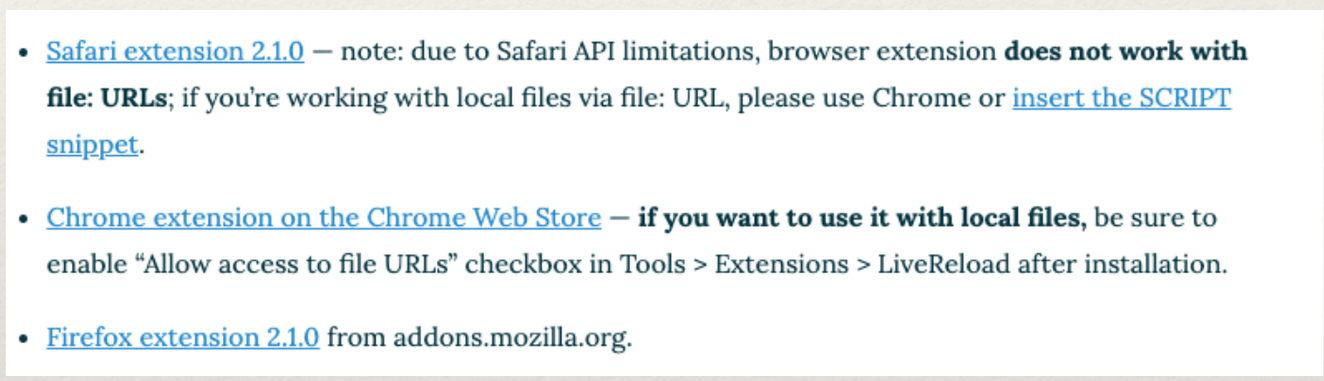
Enabling Live Reload on Browser

Another useful Spring Boot developer tools feature is **live reload**.
You can download a specific plugin for your browser from http://livereload.com/extensions/.

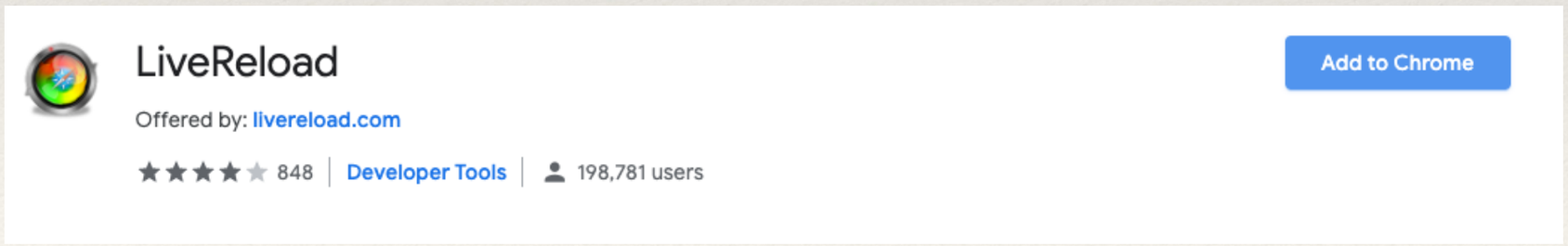
1 - Click on livereload link



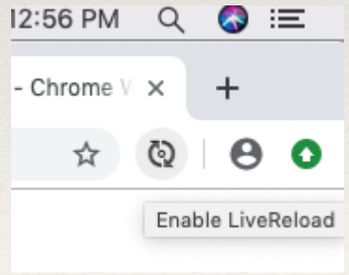
2 - Click on Chrome Ext



3 - Click on "Add to Chrome" button



4 - Icon appears



Spring Boot

Using Spring Boot Actuator

When an application is deployed in production, we want to know about the following:

- Whether a service goes down or is very slow.
- Whether any of the servers don't have sufficient free space or memory.

Spring Boot Actuator provides a number of production-ready monitoring features.

We will add Spring Boot Actuator by adding a simple dependency:

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

The actuator endpoint (`http://localhost:8080/actuator`)

You can enable all actuator URLs by configuring the property in `application.properties`:

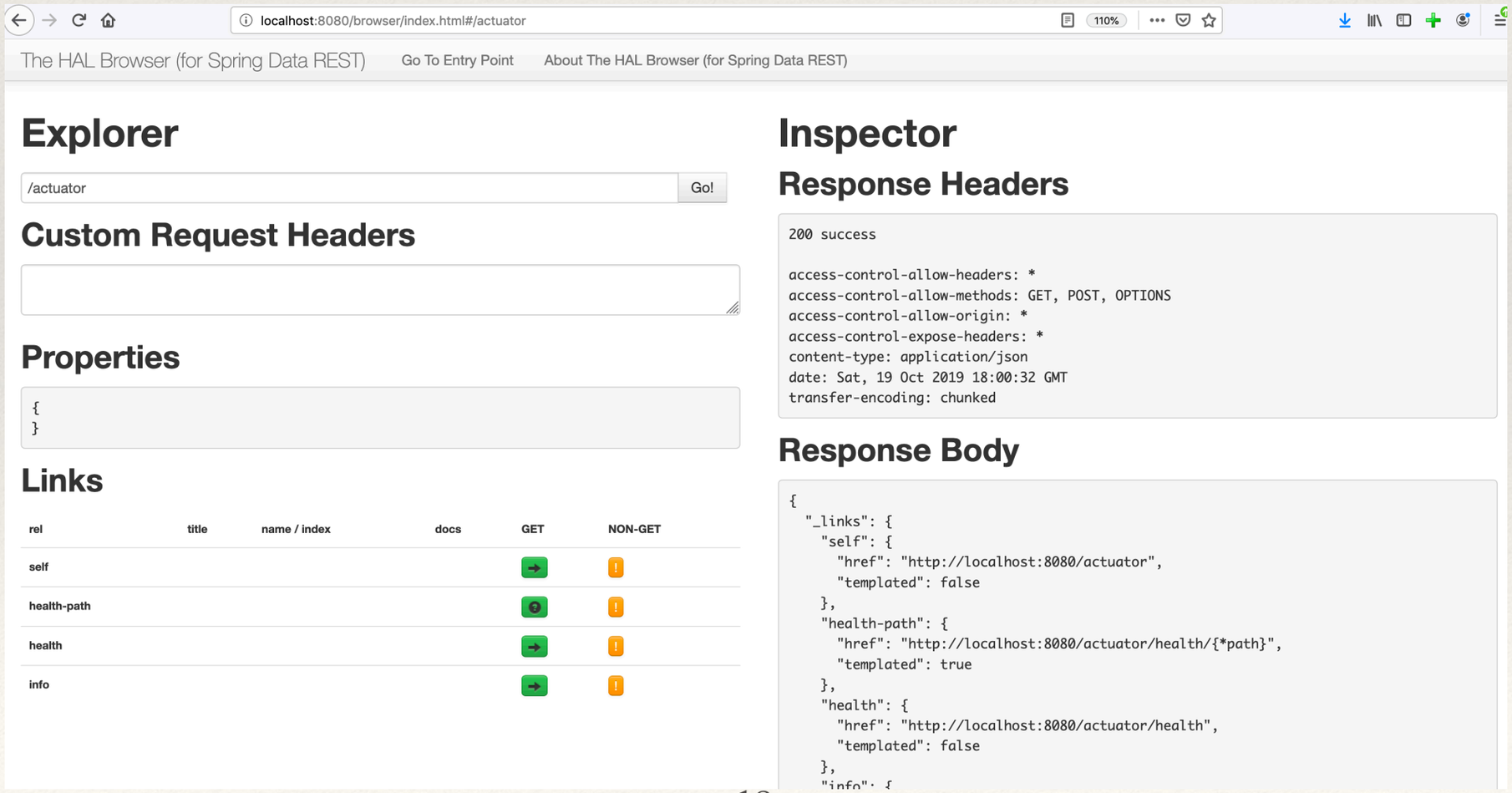
- `management.endpoints.web.exposure.include=*`

Using HAL Browser for Spring Boot Actuator

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

Exposes REST APIs around all the data that's captured from the Spring Boot application and its ENV.

The HAL BROWSER enables visual representations around the Spring Boot Actuator AP



Spring Boot

Understanding REST

Few terminologies

Server	**Service Provider.** Exposes services that can be consumed by clients.
Client	**Service Consumer.** Could be a browser or another system.
Resource	a person, an image, a video, or a product you want to sell.
Representation	A specific way in which a resource can be represented.

Spring Boot

Understanding REST

Important REST Constraints	
Client-server	• Service provider & Service consumer • Enables loose coupling
Stateless	• Each service should be stateless • No dependency on previous request being temporarily stored • Messages should be self-descriptive
Uniform Interface	• Each resource has a resource identifier. • **/users/Jack/todos/1** • Jack is the **name** of the user, and 1 is the **ID** of the todo we would want to retrieve.
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 • 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.
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.

Spring Boot

HATEOAS


```
{
    "_embedded":{
    "todos":[
            {
               "user":"Jill",
               "desc":"Learn Hibernate",
               "done":false,
               "_links":{
                 "self":{
                        "href":"http://localhost:8080/todos/1"
                  },
                 "todo":{
                     "href":"http://localhost:8080/todos/1"
                 }
             }
       }
    ]
 },
"_links":{
    "self":{
        "href":"http://localhost:8080/todos"
    },
    "profile":{
        "href":"http://localhost:8080/profile/todos"
    },
    "search":{
        "href":"http://localhost:8080/todos/search"
    }
  }
}
```

Spring Boot

Request Methods

GET	Read—retrieve details for a resource
POST	Create—create a new item or resource
PUT	Update/replace
PATCH	Update/modify a part of the resource
DELETE	Delete

Spring Boot

Designing Your First REST API

@RestController	• It 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 BasicController {

  @GetMapping("/welcome")
  public String welcome() {
    return "Hello World!!";
  }

}
```

Spring Boot

Creating REST API with JSON Response

- Create a POJO to hold the response structure
- A class with a member field called **message** and one **argument constructor** & a **getter method**

```
@GetMapping("/welcome-with-object")
public WelcomeBean welcomeWithObject() {

    return new WelcomeBean("Hello World");

}
```

Spring Boot

JSON Response with name path variable

- Path variables are used to bind values from the URI to a variable on the controller method

@GetMapping("/welcome-with-parameter/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.

Spring Boot

Creating a Todo REST API

Todo Bean	• with the ID, • the name of the user, • the description of the todo, • the todo target date, • an indicator for the completion status
TodoService	• 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
TodoController	• With **@RestController** annotation
@Autowired	• We are autowiring the todo service
@GetMapping	• to map the GET request for the "/users/{name}/todos" URI to the retrieveTodos method
@PostMapping("/users/{name}/todos")	• The @PostMapping annotation maps 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.

Spring Boot

Creating a Todo REST API

ResponseEntity.noContent().build()	• This is used to inform us that the creation of the resource failed.
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 that was created.