(http://baeldung.com)

Spring Boot Tutorial - Bootstrap una aplicación simple

Última modificación: 12 de marzo de 2018

por baeldung (http://www.baeldung.com/author/baeldung/)
(http://www.baeldung.com/author/baeldung/)

Primavera (http://www.baeldung.com/category/spring/) +

Arranque de primavera (http://www.baeldung.com/tag/spring-boot/)

1. Información general

Spring Boot es una adición obstinada, basada en la convención sobre la configuración centrada en la plataforma Spring, muy útil para comenzar con el mínimo esfuerzo y crear aplicaciones autónomas de grado de producción.

Este tutorial es un punto de partida para Boot : una forma de comenzar de manera simple, con una aplicación web básica.

Repasaremos algunas configuraciones básicas, una administración de datos rápida, de front-end y manejo de excepciones.

2. Configuración

Primero, usemos Spring Initializr (https://start.spring.io/) para generar la base de nuestro proyecto.

El proyecto generado se basa en el padre de arranque:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.3.RELEASE</version>
    <relativePath />
</parent>
```

Las dependencias iniciales van a ser bastante simples:

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

3. Configuración de la aplicación

A continuación, configuraremos una clase *principal* simple para nuestra aplicación:

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

Observe cómo usamos *@SpringBootApplication* como nuestra clase de configuración de aplicaciones principal; detrás de escena, eso es equivalente a *@Configuration* , *@EnableAutoConfiguration* y *@ComponentScan* juntos.

Finalmente, definiremos un archivo simple *application.properties* , que por ahora solo tiene una propiedad:

```
server.port=8081
```

server.port cambia el puerto del servidor de 8080 a 8081 por defecto; hay, por supuesto, muchas más propiedades Spring Boot disponibles (https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html) .

4. Simple MVC View

Ahora agreguemos una interfaz simple usando Thymeleaf.

Primero, necesitamos agregar la dependencia *spring-boot-starter-thymeleaf* a nuestro *pom.xml* :

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

Eso habilita Thymeleaf por defecto; no es necesaria ninguna configuración adicional.

Ahora podemos configurarlo en nuestra *application.properties* :

```
spring.thymeleaf.cache=false
spring.thymeleaf.enabled=true
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html

spring.application.name=Bootstrap Spring Boot
```

A continuación, definiremos un controlador simple y una página de inicio básica, con un mensaje de bienvenida:

```
@Controller
public class SimpleController {
    @Value("${spring.application.name}")
    String appName;

    @GetMapping("/")
    public String homePage(Model model) {
        model.addAttribute("appName", appName);
        return "home";
    }
}
```

Finalmente, aquí está nuestro *home.html* :

```
<html>
<head><title>Home Page</title></head>
<body>
<h1>Hello !</h1>
<p>Welcome to <span th:text="${appName}">Our App</span></p>
</body>
</html>
```

Tenga en cuenta cómo usamos una propiedad que definimos en nuestras propiedades, y luego la inyectamos para que podamos mostrarla en nuestra página de inicio.

5. Seguridad

A continuación, agreguemos seguridad a nuestra aplicación, primero incluyendo el inicio de seguridad:

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

Por ahora, con suerte se da cuenta de un patrón: la **mayoría de las bibliotecas de Spring se importan fácilmente en nuestro proyecto con el uso de simples arrancadores de arranque** .

Una vez que la dependencia *spring-boot-starter-security* en el classpath de la aplicación, Basic Autenticación está habilitada por defecto.

Y, al igual que antes, vamos a hacer una configuración simple en nuestra *application.properties* :

```
security.basic.enabled=true
security.user.name=john
security.user.password=123
```

Si no especificamos las credenciales de manera explícita, se usará un nombre de usuario predeterminado y Boot generará la contraseña aleatoriamente al inicio.

Por supuesto, Spring Security es un tema extenso y que no se cubre fácilmente en un par de líneas de configuración, así que definitivamente lo aliento a profundizar en el tema (http://www.baeldung.com/security-spring) .

6. Persistencia simple

Comencemos definiendo nuestro modelo de datos: una simple entidad de *Libro* :

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Column(nullable = false, unique = true)
    private String title;

    @Column(nullable = false)
    private String author;
}
```

Y su repositorio, haciendo un buen uso de Spring Data aquí (observe el uso de Java 8 *opcional):*

```
public interface BookRepository extends CrudRepository<Book, Long> {
    List<Book> findByTitle(String title);
    Optional<Book> findOne(long id);
}
```

Finalmente, tenemos que, por supuesto, configurar nuestra nueva capa de persistencia:

```
@EnableJpaRepositories("org.baeldung.persistence.repo")
@EntityScan("org.baeldung.persistence.model")
@SpringBootApplication
public class Application {
    ...
}
```

Tenga en cuenta que estamos usando:

- *@EnableJpaRepositories* para escanear el paquete especificado para repositorios
- *@EntityScan* para recoger nuestras entidades JPA

Para simplificar, usamos aquí una base de datos H2 en memoria, para que no tengamos dependencias externas cuando ejecutamos el proyecto.

Una vez que incluimos la dependencia de H2, **Spring Boot la detecta automáticamente y establece nuestra persistencia** sin necesidad de configuración adicional, aparte de las propiedades de la fuente de datos:

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:bootapp;DB_CLOSE_DELAY=-1
spring.datasource.username=sa
spring.datasource.password=
```

Por supuesto, al igual que la seguridad, la persistencia es un tema más amplio que este conjunto básico aquí, y uno que sin duda (http://www.baeldung.com/persistence-with-spring-series/) debe explorar más a fondo (http://www.baeldung.com/persistence-with-spring-series/) .

7. Web y el controlador

A continuación, echemos un vistazo a un nivel web, y lo comenzaremos configurando un controlador simple: el *BookController* .

Implementaremos operaciones CRUD básicas exponiendo los recursos del *Libro* con alguna validación simple:

```
@RestController
@RequestMapping("/api/books")
public class BookController {

    @Autowired
    private BookRepository bookRepository;

    @GetMapping
    public Iterable findAll() {
        return bookRepository.findAll();
    }

    @GetMapping("/title/{bookTitle}")
    public List findByTitle(@PathVariable String bookTitle) {
        return bookRepository.findByTitle(bookTitle);
    }

    @GetMapping("/{id}")
    public Book findOne(@PathVariable Long id) {
        return bookRepository.findOne(id)
          .orElseThrow(BookNotFoundException::new);
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Book create(@RequestBody Book book) {
        return bookRepository.save(book);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        bookRepository.findOne(id)
          .orElseThrow(BookNotFoundException::new);
        bookRepository.delete(id);
    }

    @PutMapping("/{id}")
    public Book updateBook(@RequestBody Book book, @PathVariable Long id) {
        if (book.getId() != id) {
          throw new BookIdMismatchException();
        }
        bookRepository.findOne(id)
          .orElseThrow(BookNotFoundException::new);
        return bookRepository.save(book);
    }
}
```

Dado que este aspecto de la aplicación es una API, hicimos uso de la anotación *@RestController* aquí, que es equivalente a *@Controller* junto con *@ResponseBody*, de modo que cada método clasifica el recurso devuelto directamente en la

respuesta HTTP.

Solo una nota que vale la pena señalar: estamos exponiendo nuestra entidad Libro como nuestro recurso externo aquí. Está bien para nuestra sencilla aplicación aquí, pero en una aplicación del mundo real, es probable que desee separar estos dos conceptos (http://www.baeldung.com/entity-to-and-from-dto-for-a-java-spring-application) .

8. Manejo de errores

Ahora que la aplicación central está lista para funcionar, centrémonos en **un mecanismo simple de manejo de errores centralizado** usando *@ControllerAdvice* :

```
@ControllerAdvice
public class RestExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler({ BookNotFoundException.class })
    protected ResponseEntity<object> handleNotFound(
      Exception ex, WebRequest request) {
        return handleExceptionInternal(ex, "Book not found",
          new HttpHeaders(), HttpStatus.NOT_FOUND, request);
    }

    @ExceptionHandler({ BookIdMismatchException.class,
      ConstraintViolationException.class,
      DataIntegrityViolationException.class })
    public ResponseEntity<object> handleBadRequest(Exception ex, WebRequest
        return handleExceptionInternal(ex, ex.getLocalizedMessage(),
          new HttpHeaders(), HttpStatus.BAD_REQUEST, request);
    }
}
```

Más allá de las excepciones estándar que manejamos aquí, también usamos una excepción personalizada:

BookNotFoundException :

```
public class BookNotFoundException extends RuntimeException {

    public BookNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    // ...
}
```

Esto debería darle una idea de lo que es posible con este mecanismo global de manejo de excepciones. Si desea ver una implementación completa, eche un vistazo al tutorial en profundidad (http://www.baeldung.com/exception-handling-for-rest-with-spring) .

Tenga en cuenta que Spring Boot también proporciona un mapeo */ error* por defecto. Podemos personalizar su vista creando un *error.html* simple :

```
<html lang="en">
<head><title>Error Occurred</title></head>
<body>
    <h1>Error Occurred!</h1>
    <b>[<span th:text="${status}">status</span>]
        <span th:text="${error}">error</span>
    </b>
    <p th:text="${message}">message</p>
</body>
</html>
```

Como la mayoría de los otros aspectos en Boot, podemos controlar eso con una propiedad simple:

```
server.error.path=/error2
```

9. Prueba

Finalmente, probemos nuestra nueva API de libros.

Inmediatamente usaremos *@SpringBootTest* para cargar el contexto de la aplicación:

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = { Application.class }, webEnvironment = WebEnviro
public class LiveTest {

    private static final String API_ROOT
      = "http://localhost:8081/api/books (http://localhost:8081/api/books)"

    @Before
    public void setUp() {
        RestAssured.authentication = preemptive().basic("john", "123");
    }
    private Book createRandomBook() {
        Book book = new Book();
        book.setTitle(randomAlphabetic(10));
        book.setAuthor(randomAlphabetic(15));
        return book;
    }

    private String createBookAsUri(Book book) {
        Response response = RestAssured.given()
          .contentType(MediaType.APPLICATION_JSON_VALUE)
          .body(book)
          .post(API_ROOT);
        return API_ROOT + "/" + response.jsonPath().get("id");
    }
}
```

Primero, podemos tratar de encontrar libros usando métodos variantes:

```
@Test
public void whenGetAllBooks_thenOK() {
    Response response = RestAssured.get(API_ROOT);

    assertEquals(HttpStatus.OK.value(), response.getStatusCode());
}

@Test
public void whenGetBooksByTitle_thenOK() {
    Book book = createRandomBook();
    createBookAsUri(book);
    Response response = RestAssured.get(
      API_ROOT + "/title/" + book.getTitle());

    assertEquals(HttpStatus.OK.value(), response.getStatusCode());
    assertTrue(response.as(List.class)
      .size() > 0);
}
@Test
public void whenGetCreatedBookById_thenOK() {
    Book book = createRandomBook();
    String location = createBookAsUri(book);
    Response response = RestAssured.get(location);

    assertEquals(HttpStatus.OK.value(), response.getStatusCode());
    assertEquals(book.getTitle(), response.jsonPath()
      .get("title"));
}

@Test
public void whenGetNotExistBookById_thenNotFound() {
    Response response = RestAssured.get(API_ROOT + "/" + randomNumeric(4));

    assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatusCode());
}
```

A continuación, probaremos la creación de un nuevo libro:

```
@Test
public void whenCreateNewBook_thenCreated() {
    Book book = createRandomBook();
    Response response = RestAssured.given()
      .contentType(MediaType.APPLICATION_JSON_VALUE)
      .body(book)
      .post(API_ROOT);

    assertEquals(HttpStatus.CREATED.value(), response.getStatusCode());
}

@Test
public void whenInvalidBook_thenError() {
    Book book = createRandomBook();
    book.setAuthor(null);
    Response response = RestAssured.given()
      .contentType(MediaType.APPLICATION_JSON_VALUE)
      .body(book)
      .post(API_ROOT);

    assertEquals(HttpStatus.BAD_REQUEST.value(), response.getStatusCode());
}
```

Actualiza un libro existente:

```
@Test
public void whenUpdateCreatedBook_thenUpdated() {
    Book book = createRandomBook();
    String location = createBookAsUri(book);
    book.setId(Long.parseLong(location.split("api/books/")[1]));
    book.setAuthor("newAuthor");
    Response response = RestAssured.given()
      .contentType(MediaType.APPLICATION_JSON_VALUE)
      .body(book)
      .put(location);

    assertEquals(HttpStatus.OK.value(), response.getStatusCode());

    response = RestAssured.get(location);

    assertEquals(HttpStatus.OK.value(), response.getStatusCode());
    assertEquals("newAuthor", response.jsonPath()
      .get("author"));
}
```

Y borra un libro:

```
@Test
public void whenDeleteCreatedBook_thenOk() {
    Book book = createRandomBook();
    String location = createBookAsUri(book);
    Response response = RestAssured.delete(location);

    assertEquals(HttpStatus.OK.value(), response.getStatusCode());

    response = RestAssured.get(location);
    assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatusCode());
}
```

10. Conclusión

Esta fue una introducción rápida pero completa a Spring Boot.

Por supuesto, apenas arañamos la superficie aquí. Hay mucho más en este marco que podemos tratar en un solo artículo introductorio.

Es exactamente por eso que no solo tenemos un solo artículo sobre Boot en el sitio (http://www.baeldung.com/tag/spring-boot/) .

El código fuente completo de nuestros ejemplos aquí es, como siempre, terminado en GitHub (https://github.com/eugenp/tutorials/tree/master/spring-boot-bootstrap) .

A simple correction needed:
"If we don't specify credentials explicitly – a default username and password will be randomly generated by Boot at startup."

Only password gets generated at startup time which is listed in the log. User name defaults to user.

+ 1 −

9 months ago

Guest

Grzegorz Piwowarek

Actually, I think this is just a comma missing 🙂

+ 0 −

9 months ago

Guest

Hendi Santika (http://hendisantika.wordpress.com)

Nice article!

+ 0 −

9 months ago

Guest

Slava Semushin

As a novice to Spring I'd like to see a full paths to all files that we're creating (main class, application.properties, both controllers and home.html). Otherwise it's possible that I put these files to non-standard location and it won't work. This part is really important because it leads to 404 errors when template is placed to a wrong location. Also AFAIR Spring Boot doesn't recommend to create a main class in the default package (https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-structuring-your-code.html#using-boot-using-the-default-package). Please, don't just post a sample of the code but also teach newbies about where and why they should be. Otherwise, we'll have more questions… Read more »

+ -1 −

9 months ago

Guest

Jeroen Wenting

all Java sources go into the package you selected when creating the application framework using Spring Initializer.
The html files go into the /resources/templates directory

+ 0 −

7 months ago

Guest

Obakeng

Slava for president! Thanks for this post. I too was getting 404-like errors due to incorrect paths set for my files. In my case, Thymeleaf threw this error: Error resolving template "home", template might not exist or might not be accessible by any of the configured Template Resolvers

My home.html was in the wrong place.

+ 1 −

7 months ago

Guest

akuma8

Nice tuto as usual. How do you run it in the embedded tomcat server ?

+ 0 −

9 months ago

Guest

Grzegorz Piwowarek

Actually, running Spring Boot jars using “java -jar” starts them by default in the embedded Tomcat. Or did you have something else in mind?

+ 0 −

9 months ago

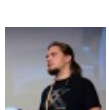
Guest

akuma8

I come from Spring MVC and try learning Spring Boot, so in a web project we usually configure 2 contexts specially the web context (with web.xml or full Java with Servlet 3) with the dispatcher, so how can I run if there isn't that configuration? Thanks

+ 0 −

9 months ago



Guest

Grzegorz Piwowarek

But here comes a very important question. Are you trying to do Spring Boot with Spring MVC or a REST-based Spring Boot application?

I am asking because Spring Boot is basically a tool that helps configuring Spring projects together. It does not force you to abandon Spring MVC.

+ 0 −

9 months ago

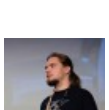
Guest

akuma8

Both, I want to mix REST and MVC, the first controller “SimpleController” is MVC based and renders the “home” page but @RestController is fully REST so where can I configure it to take that MVC part?

+ 0 −

9 months ago



Grzegorz Piwowarek

Guest

If you add the “spring-webmvc” dependency, Spring Boot will autoconfigure everything. So at this point, you can start creating views and returning their names in the @Controller-annotated classes. Also, it will run on embedded Tomcat by default so no struggling with this too. Not sure how to customize the MVC configuration, it’s been years since I did MVC last time.

+ 0 −

9 months ago

Guest

Eugen Paraschiv (http://www.baeldung.com/)

Hey @akuma8:disqus – here’s the best way to understand how you can deploy your application (including in an embedded Tomcat): http://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html (http://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html)
Hope that helps.
Cheers,
Eugen.

+ 1 −

9 months ago

Guest

alltej (https://stackoverflow.com/users/6077549/alltej)

Do you have an article regarding use/difference of Spring Hystrix vs Spring Retry (Aspect)?

+ 0 −

9 months ago

Guest

Eugen Paraschiv (http://www.baeldung.com/)

No, we haven’t published anything on that exact topic, Alltej.
Cheers,
Eugen.

+ 0 −

9 months ago

Guest

Grzegorz Piwowarek

But there are separate articles on those topics:
http://www.baeldung.com/introduction-to-hystrix (http://www.baeldung.com/introduction-to-hystrix)
http://www.baeldung.com/spring-retry (http://www.baeldung.com/spring-retry)

+ 3 −

hace 9 meses

Cargar más comentarios

CATEGORÍAS

PRIMAVERA (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST CON SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
TUTORIAL DE SPRING PERSISTENCE (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SEGURIDAD CON SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ACERCA DE

ACERCA DE BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
LOS CURSOS (HTTP://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACTO (HTTP://WWW.BAELDUNG.COM/CONTACT)