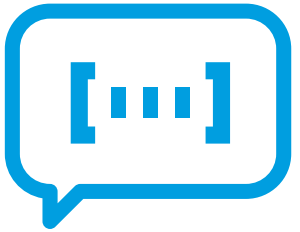
netmind
a bts company

Spring Boot

RESTful Web Services

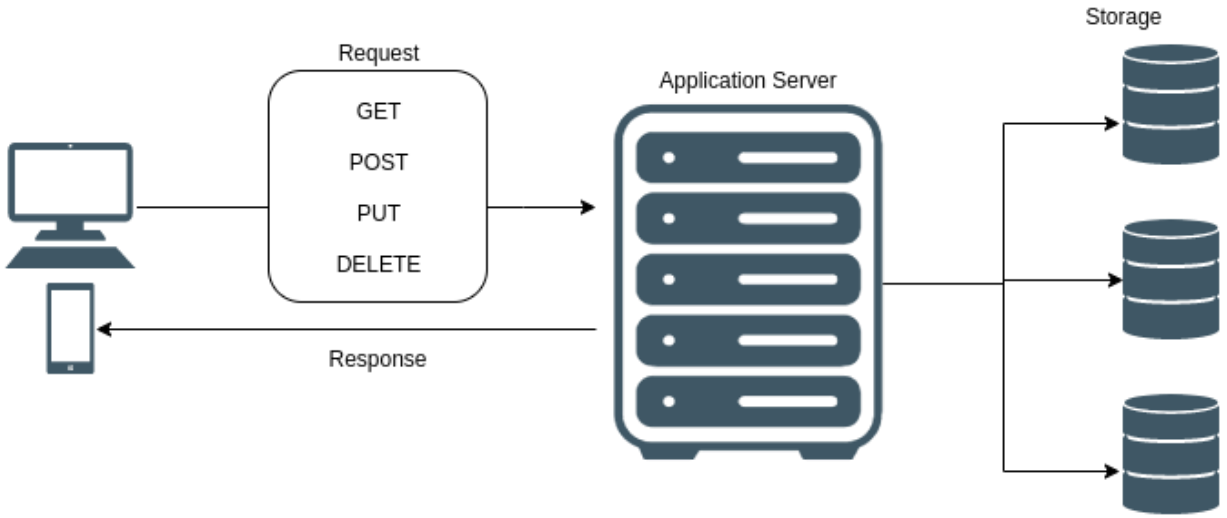
netmind
a bts company



Principios REST

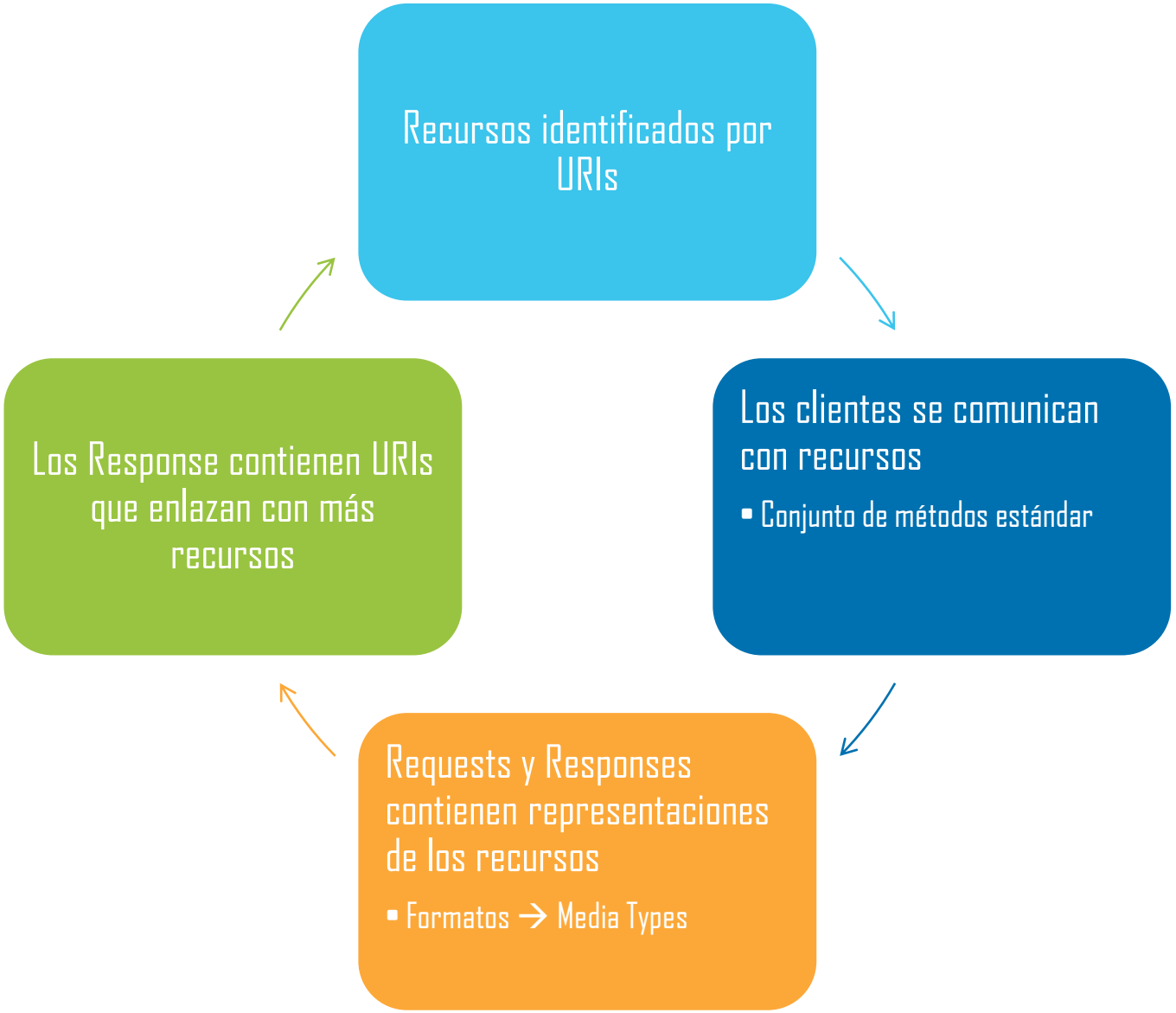
01

REST



- **REST**: Representational State Transfer
- REST es un término acuñado por **Roy Fielding** para describir un estilo de arquitectura de sistemas en red.
- NO es una tecnología o estándar
- Son un conjunto de Mejores prácticas
- Se basa en el protocolo HTTP (comandos, seguridad)
- Loosely-coupled, ligero y rápido de implementar
- Usa recursos eficientemente

Ciclo de las aplicaciones REST



Principios REST

- Dar a cada cosa un **ID**
- Conjunto de **métodos estándar**
- **Vincular** las cosas
- **Múltiples representaciones**
- Comunicaciones **SIN estado**

Dar a cada cosa un ID

- El ID de cada recurso es una URI

```
http://example.com/customers/1234
http://example.com/orders/2007/10/776654
http://example.com/products/4554
http://example.com/processes/salary-increase-234
```

- Los recursos son identificados por un ID
- Recurso == Clase Java
- Los ID se definen mediante anotación @Path

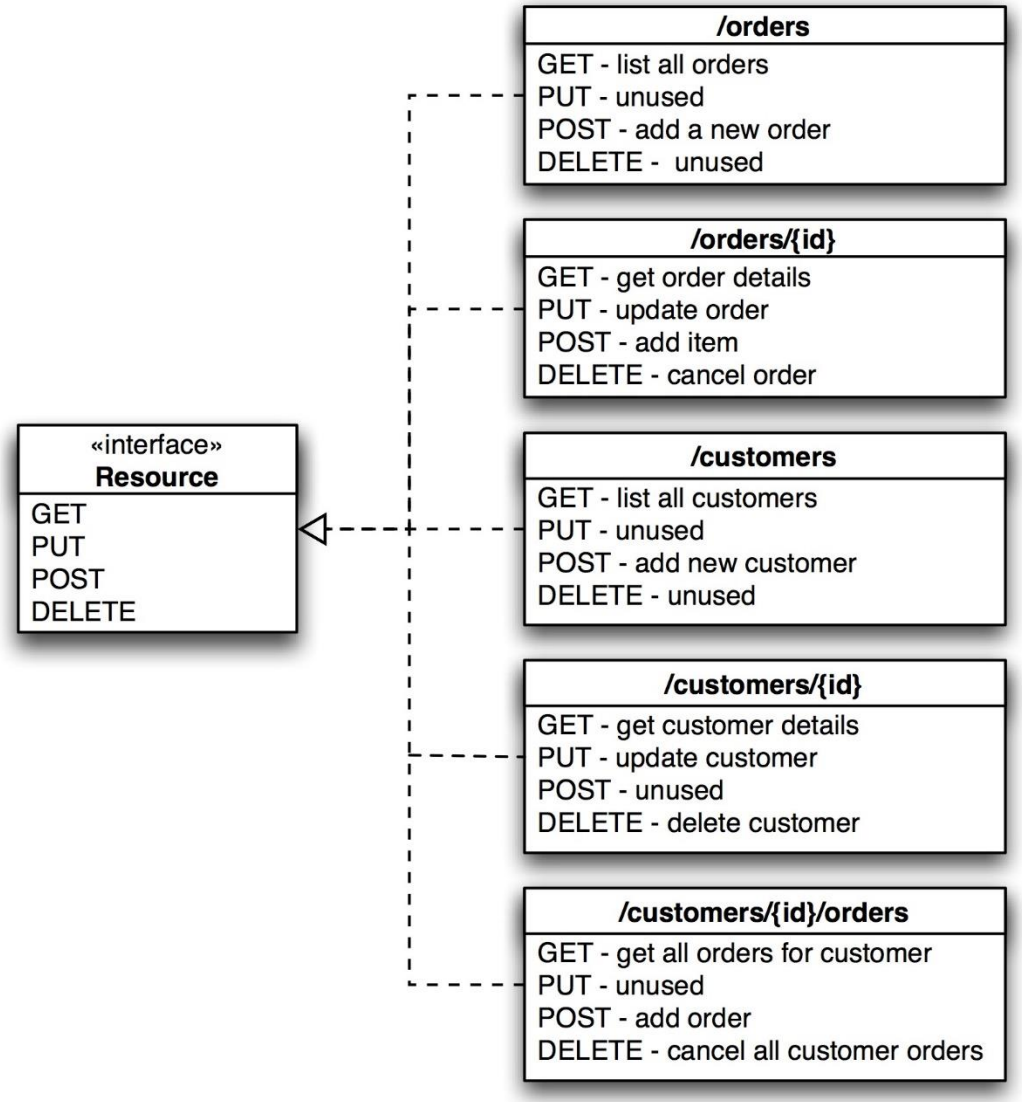
Conjunto de métodos estándar

- REST usa los comandos HTTP: PUT, DELETE, HEAD y OPTIONS

Método	Propósito	Anotación
GET	Leer (puede ser cacheado)	@GET
POST	Actualizar o crear (No idempotente)	@POST
PUT	Actualizar o crear (idempotente)	@PUT
DELETE	Borrar, eliminar	@DELETE
HEAD	GET sin Response	@HEAD
OPTIONS	Métodos soportados	@OPTIONS

- **Idempotencia**: la capacidad que tiene un ente (este caso el método) de realizar una misma operación varias veces, y obtener el mismo resultado que si solo se hiciese una vez.

Conjunto de métodos estándar



Múltiples representaciones

- Ofrece datos en una variedad de formatos
  - XML, JSON, (X)HTML
- Maximizar el alcance
- Soportar el **negociado** de contenido
- Aceptar cabecera

```
GET /foo
Accept: application/json
```

- Basada en URI

```
GET /foo.json
```

```
XML → application/properties+xml
JSON → application/properties+json
(X)HML+microformatos → application/html+xml
```

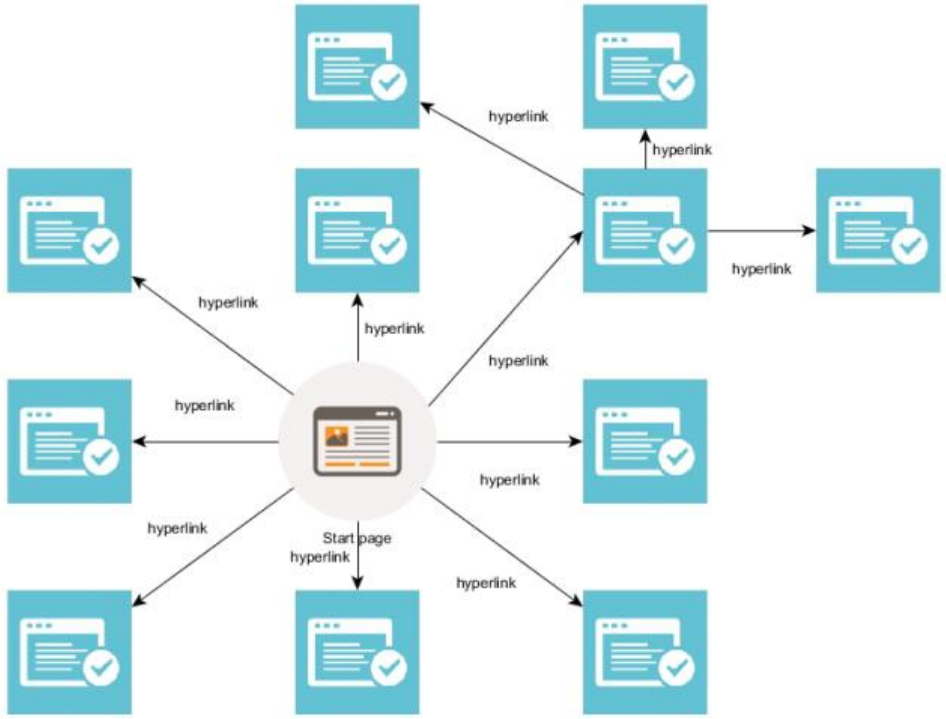
Vincular las cosas

- Los response contienen links
- Una aplicación puede interpretar los links para obtener más información.
- La belleza del enfoque de los vínculos con las URIs es que los links pueden apuntar a los recursos que son ofrecidos por una aplicación diferente.
- Concepto **HATEOAS**: Hypermedia as the engine of application state

```
{
  "stocklist": {
    "name": "ACME",
    "price": "10.00",
    "link": [
      {
        "rel": "self",
        "href": "/stock/ACME",
        "method": "get"
      },
      {
        "rel": "buy",
        "href": "/account/ACME/buy",
        "method": "post"
      },
      {
        "rel": "sell",
        "href": "/account/ACME/sell",
        "method": "post"
      }
    ]
  }
}
```

HATEOAS (Hypermedia as the engine of application state)

- "The name 'Representational State Transfer' is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."- Roy Fielding
  - Architectural Styles and the Design of Network-based Software Architectures - Chapter 6

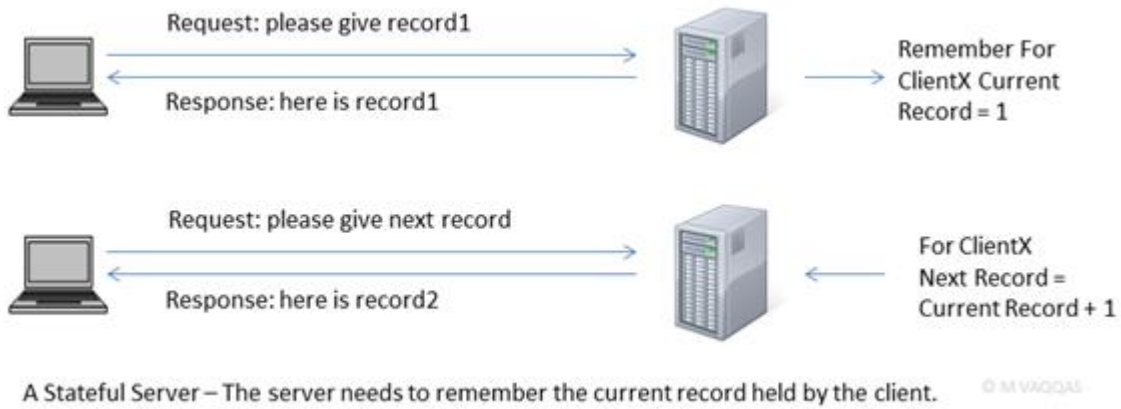


Comunicaciones SIN estado

- Los Identificadores van en la petición → independiente de la siguiente
- Todo lo que sea necesario procesar en el Request debe estar contenido en el Request
- REST exige que el estado sea transformado en estado del recurso y sea mantenido en el cliente.
- En otras palabras, **un servidor no debería guardar el estado de la comunicación de cualquiera de los clientes** que se comunican con él más allá de una petición única.
- La razón más obvia de esto es la **escalabilidad** - el número de clientes que pueden interactuar con el servidor se vería significativamente afectada si fuera necesario mantener el estado del cliente.

Diseño Sin Estado

- Sin REST



A Stateful Server – The server needs to remember the current record held by the client.

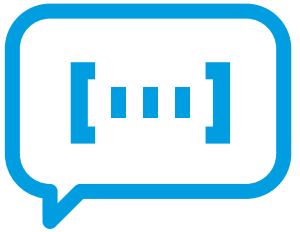
- Con REST

A Stateless Server– The server does not need to remember which record is being held by the client

Principios REST extendidos

1. Identificar todas las entidades conceptuales que deseamos exponer como servicios.
2. Crear una URL para cada recurso.
3. Categorizar nuestros recursos según si los clientes pueden recibir una representación del recurso (usando HTTP GET), o si los clientes pueden modificar (agregar a) el recurso usando HTTP POST, PUT y/o DELETE).
4. Todos los recursos accesibles a través de HTTP GET deben estar libres de efectos secundarios. Es decir, el recurso solo debe devolver una representación del recurso. La invocación del recurso no debería resultar en la modificación del recurso.
5. Colocar hipervínculos dentro de las representaciones de recursos para permitir que los clientes profundicen en busca de más información y/u obtengan información relacionada.
6. Diseñar para revelar datos gradualmente. No revelar todo en un solo documento de respuesta. Proporcionar hipervínculos para obtener más detalles.
7. Especificar el formato de los datos de respuesta mediante un esquema (DTD, W3C Schema, RelaxNG o Schematron). Para aquellos servicios que requieren un POST o PUT, también proporcione un esquema para especificar el formato de la respuesta.
8. Describit cómo se invocarán nuestros servicios mediante un documento **swagger** o simplemente un documento HTML.

netmind
a bts company



Diseño de APIs REST

02

API REST

- API es que es la abreviatura de **Application Programming Interface**, o Interfaz de Programación de Aplicaciones
- Una API consigue que los desarrolladores interactúen con los datos de la aplicación de un modo planificado y ordenado.
- Una API REST es una interfaz de acceso a los servicios de una aplicación web.
- Diseñar una API es un proceso que hay que seguir con cuidado ya que será usada por aplicaciones clientes y por terceras partes.
- El objetivo del diseño es lograr una API que sea **fácil de usar**, **fácil de adoptar** y lo suficientemente **flexible para implementarla** en nuestras propias interfaces de usuario.

Requisitos fundamentales para una API

- ✓ Debe utilizar estándares web donde tengan sentido
- ✓ Debe ser amigable para el desarrollador y ser explorable mediante una barra de direcciones del navegador
- ✓ Debe ser sencilla, intuitiva y consistente para que la adopción no sólo sea fácil, sino también agradable
- ✓ Debe proporcionar suficiente flexibilidad para potenciar mayoritariamente la UI
- ✓ Debe ser eficiente, manteniendo el equilibrio con los demás requisitos
- ✓ Una API es una interfaz de usuario (UI) para un desarrollador – al igual que cualquier UI, es importante asegurar que la experiencia del usuario esté pensada cuidadosamente!

Uso correcto de URIs

- Las URL, Uniform Resource Locator , son un tipo de URI, Uniform Resource Identifier, que además de permitir **identificar de forma única el recurso**, nos permite localizarlo para poder acceder a él o compartir su ubicación.

- Una URL se estructura de la siguiente forma:

```
{protocolo}://{dominio o hostname}[:puerto (opcional)]/{ruta del recurso}?{consulta de filtrado}
```

Reglas básicas para URI de un recurso

Regla	Incorrecto	Correcto
Los nombres de URI no deben implicar una acción, por lo tanto debe evitarse usar verbos en ellos.	/facturas/234/editar	/facturas/234 para editar, borrar, consultar
Deben ser únicas, no debemos tener más de una URI para identificar un mismo recurso.		
Deben ser independiente de formato.	/facturas/234.pdf	/facturas/234 para pdf, epub, txt, xml o json
Deben mantener una jerarquía lógica.	/facturas/234/cliente/007	/clientes/007/facturas/234
Los filtrados de información de un recurso no se hacen en la URI.	/facturas/orden/desc/fecha-desde/2007/pagina/2	/facturas?fecha-desde=2007&orden=DESC&pagina=2

Usa acciones y URLs RESTful

- Estos recursos son manipulados usando peticiones HTTP donde el método (GET, POST, PUT, PATCH, DELETE) tienen un significado específico.
- Ejemplos:
  - **GET /facturas** Nos permite acceder al listado de facturas
  - **POST /facturas** Nos permite crear una factura nueva
  - **GET /facturas/123** Nos permite acceder al detalle de una factura
  - **PUT /facturas/123** Nos permite editar la factura, sustituyendo la totalidad de la información anterior por la nueva.
  - **DELETE /facturas/123** Nos permite eliminar la factura
  - **PATCH /facturas/123** Nos permite modificar cierta información de la factura, como el número o la fecha de la misma.

Códigos de estado

- Un error común es no devolver el código de estado que indique la respuesta.
  - https://es.wikipedia.org/wiki/Anexo:C%C3%B3digos_de_estado_HTTP

Incorrecto: el código es 200, pero en el cuerpo de respuesta indicamos un error

```
Petición
========
PUT /facturas/123

Respuesta
=========
Status Code 200
Content:
{
  success: false,
  code:    734,
  error:   "datos insuficientes"
}
```

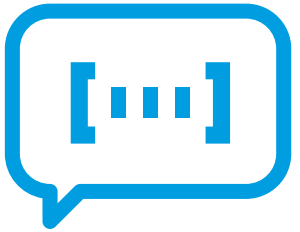
Correcto: el código es 400, que se corresponde con el error del cuerpo

```
Petición
========
PUT /facturas/123

Respuesta
=========
Status Code 400
Content:
{
  message: "se debe especificar
un id de cliente para la factura"
}
```

Más recomendaciones

- Usa SSL en todos lados, sin excepciones
- Una API es tan buena como lo es su documentación – por lo tanto realiza una buena documentación
- HATEOAS todavía no es muy práctico
- Usa JSON donde sea posible, XML sólo si tienes la obligación
- Deberías usar camelCase con JSON, pero snake_case es un 20% más fácil de leer
- Considera usar JSON para cuerpos de peticiones POST, PUT y PATCH
- Usa autenticación basada en tokens, transportado en OAuth2 donde se necesite delegación
- Utilizar efectivamente los códigos de error HTTP

netmind
a bts company



Implementación REST con SpringBoot

03

Dependencias

- Spring Boot brinda un muy buen soporte para crear servicios web RESTful.

- Deberemos agregar la dependencia web Spring Boot Starter en el archivo de configuración de compilación **pom.xml**.

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

Soporte Spring para REST

- Spring facilita la creación de servicios REST con el componente MVC
- Los mecanismos de los que dispone incluyen:
  - **@RequestMapping** mapea URIs RESTful a controladores estándar
    - https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html
  - **@PathVariable** soporta plantillas URI
  - **@RequestBody** enlaza los datos recibidos por HTTP a objetos Java
  - **@ResponseBody** enlaza los return a el cuerpo del response HTTP
  - **Soporte de métodos HTTP**: GET, POST, DELETE, PUT
  - **Vistas** que soportan representaciones XML y JSON en la respuesta

Path templates y @PathVariable

- Spring permite definir rutas con variables usando {<nombre_varianble>}
  - Por ejemplo:

```
@RequestMapping("/customers/{id}")
```

- Esta variable la podremos asociar a un parámetro del método que implementa el endpoint usando @PathVariable

```
@RequestMapping("/customers/{id}")
public String getCustomer(@PathVariable Long id) {
        /* … */
}
```

- Por tanto si hacemos una petición a la siguiente ruta

```
/customers/234
```

- La variable **id** va

Path templates y @PathVariable

- Se pueden añadir las variables que sean necesarias en el path
- Para manipularlas, crearemos las path variables correspondientes

```
@RequestMapping("/fname/{fname}/lname/{lname}")
public String getCustomer(
     @PathVariable("fname") String firstName,
     @PathVariable("lname") String lastName
){
     …
}
```

@ResponseBody

- Indica que el valor devuelto contiene la respuesta en sí misma.
- Los datos devueltos son convertidos a una representación que el cliente puede aceptar.

```
@RequestMapping("/customers/{id}")
@ResponseBody
public Customer getCustomer(@PathVariable Long id){
         Customer result = … // Find a Customer (not shown)
         return result;
}
```

- Para ello el cliente debe enviar en la cabecera Accept Header el MIME aceptado. Por ejemplo:
  - text/xml
  - application/json

@RestController

- Combina @Controller y @ResonseBody
- Esto hace innecesario añadir @ResponseBody
- Útil para clases REST puras

```
@RestController
@RequestMapping("/customers")
public class CustomerResource {

     @RequestMapping("/{id}") // @ResponseBody not needed
     public String getCustomer(@PathVariable Long id) {
          Customer result = … // Find a Customer (not shown)
          return result.toString();
     }

}
```

@RequestBody

- Se usa para definir el tipo de contenido del cuerpo de la solicitud.

```
public ResponseEntity<Object> createProduct(@RequestBody Product product) {
}
```

@RequestParam

- Se usa para leer los parámetros de solicitud de la URL de solicitud.
  - Por defecto, es un parámetro obligatorio.
  - Podemos establecer el valor predeterminado.

```
public ResponseEntity<Object> getProduct(
   @RequestParam(value = "name", required = false, defaultValue = "honey") String name) {
}
```

GET API

- El método de solicitud HTTP predeterminado es GET.
- Este método no requiere ningún cuerpo en el request.
- Se puede enviar parámetros de solicitud y variables de ruta para definir la URL personalizada o dinámica.

```
@RequestMapping(value = "/products")
public ResponseEntity<Object> getProduct() {
      return new ResponseEntity<>(productRepo.values(), HttpStatus.OK);
}
```

POST API

- La solicitud HTTP POST se utiliza para crear un recurso.
- Este método contiene el cuerpo de solicitud.
- Podemos enviar parámetros de solicitud y variables de ruta para definir la URL personalizada o dinámica.

```
@RequestMapping(value = "/products", method = RequestMethod.POST)
public ResponseEntity<Object> createProduct(@RequestBody Product product) {
      productRepo.put(product.getId(), product);
      return new ResponseEntity<>("Product is created successfully", HttpStatus.CREATED);
}
```

PUT API

- La solicitud HTTP PUT se utiliza para actualizar el recurso existente.
- Este método contiene un cuerpo de solicitud.
- Podemos enviar parámetros de solicitud y variables de ruta para definir la URL personalizada o dinámica.

```
@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)
public ResponseEntity<Object> updateProduct(
          @PathVariable("id") String id, @RequestBody Product product
) {
      productRepo.remove(id);
      product.setId(id);
      productRepo.put(id, product);
      return new ResponseEntity<>("Product is updated successsfully", HttpStatus.OK);
}
```

DELETE API

- La solicitud de eliminación HTTP se utiliza para eliminar el recurso existente.
- Este método no contiene ningún cuerpo de solicitud.
- Podemos enviar parámetros de solicitud y variables de ruta para definir la URL personalizada o dinámica.

```
@RequestMapping(value = "/products/{id}", method = RequestMethod.DELETE)
public ResponseEntity<Object> delete(@PathVariable("id") String id) {
      productRepo.remove(id);
      return new ResponseEntity<>("Product is deleted successsfully", HttpStatus.OK);
}
```

Atajos para RequestMapping

- Spring actualmente admite cinco tipos de anotaciones integradas para manejar diferentes tipos de métodos de solicitud HTTP entrantes que son GET, POST, PUT, DELETE y PATCH.
- Estas anotaciones son:
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
  - @PatchMapping

```
@GetMapping("/get/{id}")
public @ResponseBody ResponseEntity<String>
  getById(@PathVariable String id) {
    return new ResponseEntity<String>("GET Response : "+ id, HttpStatus.OK);
}
```

Detalles de RequestMapping

- **@RequestMapping con Class**: podemos usarla con la definición de clase para crear el URI base.

```
@Controller
@RequestMapping("/home")
public class HomeController {
}
```

- **@RequestMapping con múltiples URI**: podemos usar un solo método para gestionar múltiples URI.

```
@RequestMapping(value={"/method1","/method1/second"})
public String method1(){
        return "method1";
}
```

- **@RequestMapping con headers**: podemos especificar las cabeceras que deben estar presentes para invocar el método del controlador.

```
@RequestMapping(value="/method5", headers={"name=pankaj", "id=1"})
public String method5(){
        return "method5";
}
```

Detalles de RequestMapping

- **@RequestMapping con varios request methods**: a veces queremos realizar diferentes operaciones según el método HTTP utilizado, aunque el URI de la solicitud sigue siendo el mismo.

```
@RequestMapping(value="/method3",
method={RequestMethod.POST,RequestMethod.GET})
public String method3(){
          return "method3";
}
```

- **Método alternativo**: podemos crear un método alternativo para la clase de controlador para asegurarnos de que estamos capturando todas las solicitudes del cliente aunque no haya métodos de controlador coincidentes.
  - Es útil para enviar páginas de respuesta 404 personalizadas a los usuarios cuando no hay métodos de manejo para la solicitud.

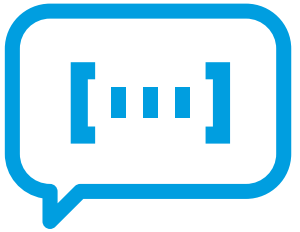
```
@RequestMapping("*")
public String fallbackMethod(){
          return "fallback method";
}
```

Negociado de contenidos

- **@RequestMapping con Produce y Consumes**: podemos usar el encabezado **Content-Type** y **Accept** para definir el contenido de la solicitud y cuál es el mensaje MIME que quiere como respuesta.
- Para mayor claridad, @RequestMapping proporciona variables de producción y consumo donde podemos especificar el tipo de contenido de la solicitud para el cual se invocará el método y el tipo de contenido de la respuesta.

```
@RequestMapping(value="/method6",
produces={"application/json","application/xml"},
consumes="text/html“)
public String method6(){
          return "method6";
}
```

- También se puede usar la clase MediaType
  - https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/MediaType.html
  - JSON: MediaType.APPLICATION_JSON
  - XML: MediaType.TEXT_XML



Manejo de excepciones

04

Manejo de excepciones

- Gestionar excepciones y errores en las API y enviar la respuesta adecuada al cliente es esencial para las aplicaciones empresariales.
- Las siguientes anotaciones se verán implicadas
- **@ControllerAdvice** es una anotación para manejar las excepciones globalmente.
- **@ExceptionHandler** es una anotación que se usa para manejar las excepciones específicas y enviar las respuestas personalizadas al cliente.

```
@ControllerAdvice
public class ProductExceptionController {

   @ExceptionHandler(value = ProductNotfoundException.class)
   public ResponseEntity<Object> exception(ProductNotfoundException exception) {
      return new ResponseEntity<>("Product not found", HttpStatus.NOT_FOUND);
   }

}
```

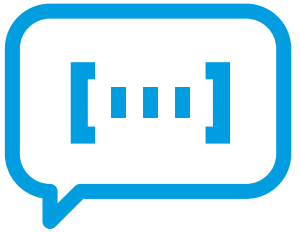
Manejo de excepciones

- Definir una clase que extienda la clase RuntimeException.

```
public class ProductNotfoundException extends RuntimeException {
   private static final long serialVersionUID = 1L;
}
```

- Usar la excepción en el request mapping

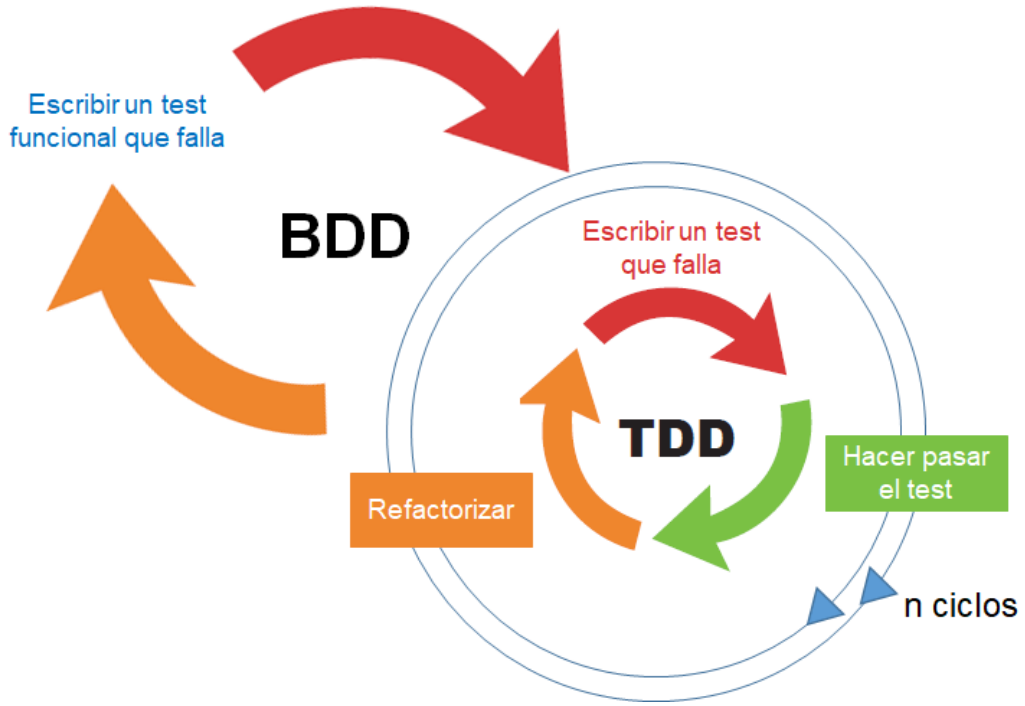
```
@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)
public ResponseEntity<Object> updateProduct() {
   throw new ProductNotfoundException();
}
```



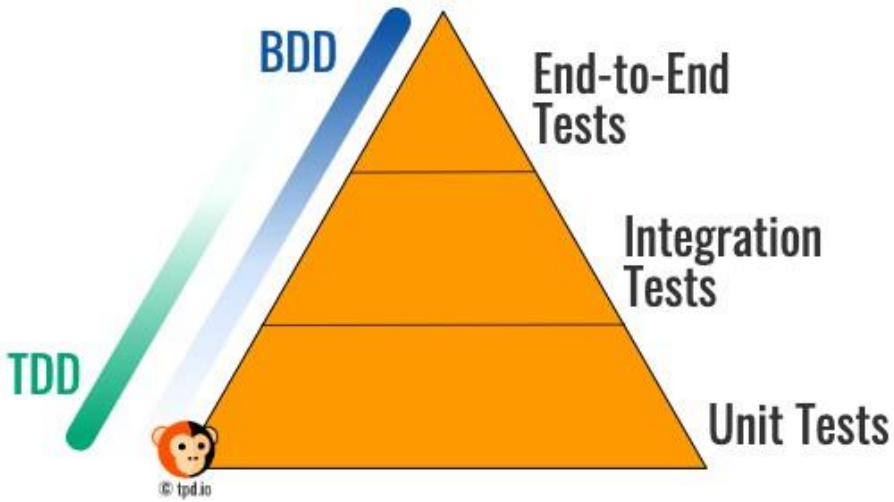
Testing en Spring Boot

05

BDD y TDD



- El Desarrollo guiado por pruebas (**TDD**: Test Driven Development) y el Desarrollo guiado por comportamiento (**BDD**: Behavior Driven Development) son metodologías complementarias que tienen como objetivo asegurar la calidad del software desde la fuente.
- Hemos visto que BDD puede servir para definir los servicios, basándonos en su comportamiento.



- Spring Boot permite provee herramientas para usar estos métodos incluyendo test automáticos en el delivery de nuestras aplicaciones.

Dependencias Maven

- Spring-boot-starter-test es la dependencia principal que contiene la mayoría de los elementos necesarios para nuestras pruebas.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <version>2.5.0</version>
</dependency>
```

- **Junit 4:** A partir de Spring Boot 2.4, el antiguo motor de JUnit se eliminó de spring-boot-starter-test. Si aún queremos usar JUnit 4, debemos agregar la siguiente dependencia:

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Pruebas de integración con @SpringBootTest

- Las pruebas de integración se enfocan en integrar diferentes capas de la aplicación.
- Idealmente, deberíamos mantener las pruebas de integración separadas de las pruebas unitarias y no deberíamos ejecutarlas juntas.
- Podemos hacer esto usando un perfil diferente para ejecutar solo las pruebas de integración.
- Las razones para hacer esto es que:
  - las pruebas de integración consumen mucho tiempo y
  - pueden necesitar una base de datos real para ejecutarse.

Pruebas de integración con @SpringBootTest

- La anotación @SpringBootTest es útil cuando necesitamos arrancar todo el contenedor. La anotación funciona creando el **ApplicationContext** que se utilizará en nuestras pruebas.
  - Lo que significa que podemos @Autowire cualquier bean que haya sido recogido por el escaneo de componentes en nuestra prueba:
- Podemos usar el atributo **webEnvironment** de @SpringBootTest para configurar nuestro entorno de tiempo de ejecución; se usa WebEnvironment.MOCK para que el contenedor funcione en un entorno de servlet simulado.
- La anotación **@TestPropertySource** ayuda a configurar las ubicaciones de los archivos de propiedades específicos de nuestras pruebas.
  - Tener en cuenta que el archivo de propiedades cargado con @TestPropertySource anulará el archivo application.properties existente.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK,
        classes = ProductsServiceApplication.class)
@AutoConfigureMockMvc
@TestPropertySource(locations = "classpath:application-integrationtest.properties")
public class ProductServiceControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private ProductsRepository repository;

    // test cases here
}
```

Pruebas de integración con @SpringBootTest

- Los casos de prueba para las pruebas de integración pueden parecerse a las pruebas unitarias de la capa del controlador:

```
@Test
public void givenProducts_whenGetProducts_thenStatus200() throws Exception {
    Product nuevoProd = new Product();
    nuevoProd.setName("Nuevo Prod");
    repository.save(nuevoProd);

    mvc.perform(get("/products").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$[0].name", is("Nuevo Prod")));
}
```

- La diferencia con las pruebas unitarias de la capa del controlador es que aquí no se simula nada y se ejecutan escenarios de extremo a extremo.

Configuración de prueba con @TestConfiguration

- Es posible que deseemos evitar el arranque del contexto de la aplicación real y usar una configuración de prueba especial. Podemos lograr esto con la anotación @TestConfiguration .
- Hay dos formas de usar la anotación

(1). Una **clase interna estática** en la misma clase de prueba donde queremos @Autowire the bean:

```
@ExtendWith(SpringExtension.class)
public class ProductServiceImplIntegrationTest {

    @TestConfiguration
    static class ProductServiceImplTestContextConfiguration {
        @Bean
        public ProductsService productsService() {
           return new EmployeeService() {
                // implement methods
            };

        }
    }
}
```

Configuración de prueba con @TestConfiguration

(2). Crear una **clase de configuración de prueba** separada:

```
@TestConfiguration
public class ProductServiceImplTestContextConfiguration {
    @Bean
    public ProductsService productsService() {
       return new EmployeeService() {
            // implement methods
        };

    }
}
```

- Las clases de configuración anotadas con @TestConfiguration están excluidas del análisis de componentes, por lo tanto, debemos importarlas explícitamente en cada prueba en la que queramos hacer @Autowire .

```
@ExtendWith(SpringExtension.class)
@Import(ProductServiceImplTestContextConfiguration.class)
public class ProductServiceImplIntegrationTest {

    @Autowired
    private ProductsService productsService;

    // remaining class code
}
```

Mocking con con @MockBean

- Nuestro código de capa de Servicio usualmente depende de nuestro Repositorio.
- Sin embargo, para probar la capa de servicio , no necesitamos saber cómo se implementa la capa de persistencia ni preocuparnos por ella. Idealmente, deberíamos poder escribir y probar nuestro código de capa de servicio sin cableado en nuestra capa de persistencia completa.
- Para lograr esto, podemos usar el soporte de simulación proporcionado por Spring Boot Test.

```
@ExtendWith(SpringExtension.class)
public class ProductServiceImplIntegrationTest {

    @TestConfiguration
    static class ProductServiceImplTestContextConfiguration {
        @Bean
        public ProductsService productsService() {
            return new ProductsService();
        }
    }

    @Autowired
    private ProductsService productsService;

    @MockBean
    private ProductsRepository productsRepository;

    // write test cases here

}
```

Mocking con con @MockBean

- @MockBean crea un simulacro para el EmployeeRepository , que se puede usar para omitir la llamada al EmployeeRepository real.

```
public void setUp() {
    List<Product> products = Arrays.asList(
        new Product(1L,"Fake product")
    );
    Mockito.when(productsRepository.findByNameContaining("Fake"))
            .thenReturn(products);
}
```

- El caso de prueba será más simple:

```
@Test
public void whenValidText_thenProductsShouldBeFound() {
    String text = "Fake";
    List<Product> found = productsService.
        getProductsByText(text);

    assertThat(found).isNotEmpty();
    assertThat(found.get(0).getName()).contains("Fake");
}
```

Pruebas de integración con @DataJpaTest

- **@ExtendWith(SpringExtension.class)** proporciona un puente entre las funciones de prueba de Spring Boot y JUnit.
  - Siempre que estemos usando cualquier característica de prueba de Spring Boot en nuestras pruebas JUnit, se requerirá esta anotación.
- **@DataJpaTest** proporciona una configuración estándar necesaria para probar la capa de persistencia:
  - configurar H2, una base de datos en memoria
  - configurar Hibernate, Spring Data y DataSource
  - realizando un @EntityScan
  - activar el registro de SQL
- Para llevar a cabo operaciones de base de datos, necesitamos algunos registros que ya estén en nuestra base de datos.
  - Para configurar estos datos, podemos usar **TestEntityManager**.
  - Spring Boot TestEntityManager es una alternativa al JPA EntityManager estándar que proporciona métodos comúnmente utilizados al escribir pruebas.

Pruebas de integración con @DataJpaTest

```
@ExtendWith(SpringExtension.class)
@DataJpaTest
public class ProductRepositoryIntegrationTest {
    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private ProductsRepository productsRepository;

    @Test
    public void whenFindByName_thenReturnProduct() {
        // given
        Product aProduct = new Product(null, "Fake Product");
        entityManager.persist(aProduct);
        entityManager.flush();

        // when
        Product found = productsRepository.findByName(aProduct.getName());

        // then
        assertThat(found.getName())
                .isEqualTo(aProduct.getName());
    }

}
```

Pruebas unitarias con @WebMvcTest

- Para probar los controladores , podemos usar **@WebMvcTest**.
  - Configurará automáticamente la infraestructura Spring MVC para nuestras pruebas unitarias.
- En la mayoría de los casos, @WebMvcTest se limitará a arrancar un solo controlador.
- También podemos usarlo junto con @MockBean para proporcionar implementaciones simuladas para cualquier dependencia requerida.
- @WebMvcTest también configura automáticamente MockMvc , que ofrece una forma poderosa de probar fácilmente los controladores MVC sin iniciar un servidor HTTP completo.

Pruebas unitarias con @WebMvcTest

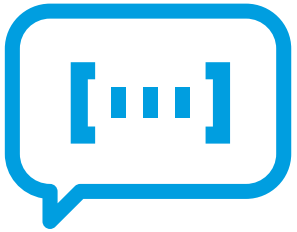
```
@ExtendWith(SpringExtension.class)
@WebMvcTest(ProductServiceController.class)
public class ProductServiceControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private ProductsRepository repository;

    @Test
    public void givenProducts_whenGetProducts_thenReturnJsonArray() throws Exception {

        Product aProduct = new Product(1L, "Fake product");

        List<Product> allProducts = Arrays.asList(aProduct);

        given(repository.findAll()).willReturn(allProducts);

        mvc.perform(get("/products")
                        .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(1)))
                .andExpect(jsonPath("$[0].name", is(aProduct.getName())));
    }

}
```

Pruebas configuradas automáticamente

- Además de las anotaciones anteriores, algunas anotaciones ampliamente utilizadas:
  - **@WebFluxTest**: probar los controladores Spring WebFlux. A menudo se usa junto con @MockBean para proporcionar implementaciones simuladas para las dependencias requeridas.
  - **@JdbcTest**: probar aplicaciones JPA, pero es para pruebas que solo requieren un DataSource. La anotación configura una base de datos incrustada en memoria y un JdbcTemplate.
  - **@JooqTest**: pruebas relacionadas con jOOQ. Configura un DSLContext.
  - **@DataMongoTest**: para probar aplicaciones MongoDB. De manera predeterminada, configura un MongoDB incrustado en memoria si el controlador está disponible a través de dependencias, configura un MongoTemplate, busca clases de @Document y configura repositorios de Spring Data MongoDB.
  - **@DataRedisTest**: prueba de aplicaciones de Redis. Busca clases de @RedisHash y configura repositorios Spring Data Redis de forma predeterminada.
  - **@DataLdapTest**: configura un LDAP incrustado en la memoria (si está disponible), configura un LdapTemplate , busca clases de @Entry y configura repositorios Spring Data LDAP de manera predeterminada.
  - **@RestClientTest**: probar clientes REST. Configura automáticamente diferentes dependencias, como la compatibilidad con Jackson, GSON y Jsonb; configura un RestTemplateBuilder; y agrega soporte para MockRestServiceServer de forma predeterminada.
  - **@JsonTest**: Inicializa el contexto de la aplicación Spring solo con los beans necesarios para probar la serialización JSON.

netmind
a bts company

06



CORS

Habilitar CORS

- Cross-Origin Resource Sharing (CORS) es un concepto de seguridad que permite restringir los recursos implementados en los navegadores web.
  - Evita que el código JavaScript produzca o consuma las solicitudes contra un origen diferente.
- Los servicios web RESTful deben admitir el uso compartido de recursos de origen cruzado.

Habilitar CORS en el método del controlador

- Necesitamos establecer los orígenes para el servicio web RESTful usando la anotación @CrossOrigin para el método del controlador o de método.

```
@RequestMapping(value = "/products")
@CrossOrigin(origins = "http://localhost:8080")

public ResponseEntity<Object> getProduct() {
   return null;
}
```

Probar CORS

- Usa el siguiente código Javascript para probar:

```
const test_cors_url = (url) => {

    fetch(url).then(response => response.json())
      .then(json => {
        console.log('Allowed', json);
})
      .catch(error => {
        console.error('Not allowed - Error fetching:', error);
      });

}
```

Habilitar CORS

Configuración global de CORS

- Necesitamos definir un @Bean de configuración para establecer el soporte de configuración de CORS global.

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

   @Bean
   public WebMvcConfigurer corsConfigurer() {
      return new WebMvcConfigurerAdapter() {
         @Override
         public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/products").allowedOrigins("http://localhost:8080");
         }
      };
   }
}
```

07

Validación

Validación con Spring boot

- Cuando se trata de validar la entrada del usuario, Spring Boot brinda un sólido soporte para esta tarea común, pero crítica, desde el primer momento.
  - Para la validación se usa la Jakarta Bean Validation
    - https://beanvalidation.org/2.0/

- A partir de Boot 2.3, también debemos agregar explícitamente la dependencia **spring-boot-starter-validation**:

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

Anotaciones de validación comunes

- Algunas de las anotaciones de validación más comunes son:
  - **@NotNull**: indica que un campo no debe ser nulo.
  - **@NotEmpty**: indica que un campo de lista no debe estar vacío.
  - **@NotBlank**: indica que un campo de cadena no debe ser la cadena vacía (es decir, debe tener al menos un carácter).
  - **@Miny @Max**: indica que un campo numérico sólo es válido cuando su valor está por encima o por debajo de un determinado valor.
  - **@Pattern**: indica que un campo de cadena solo es válido cuando coincide con una determinada expresión regular.
  - **@Email**: indica que un campo de cadena debe ser una dirección de correo electrónico válida.

Clase de dominio

Con las dependencias de nuestro proyecto ya establecidas, a continuación debemos definir una clase de entidad JPA de ejemplo, cuyo rol será únicamente modelar a los usuarios.

```
@Entity
public class User {

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

    @NotBlank(message = "Name is mandatory")
    private String name;

    @NotBlank(message = "Email is mandatory")
    private String email;

    // standard constructors / setters / getters / toString

}
```

Validando el Request Body

- Nos permitará obtener los valores asignados a los campos restringidos de nuestro modelo.

```
@RestController
public class UserController {

    @PostMapping("/users")
    ResponseEntity<String> addUser(@Valid @RequestBody User user) {
        // persisting the user
        return ResponseEntity.ok("User is valid");
    }

    // standard constructors / other methods

}
```

- La parte más relevante es el uso de la anotación **@Valid**.
- Cuando Spring Boot encuentra un argumento anotado con @Valid, inicia automáticamente la implementación predeterminada de JSR 380 (Hibernate Validator) y valida el argumento.
- Cuando el argumento de destino no pasa la validación, Spring Boot genera una excepción **MethodArgumentNotValidException**.

Configuración de mensajes

- Spring Boot proporciona algunas propiedades con las que podemos agregar el mensaje de excepción, la clase de excepción o incluso la traza de errores como respuesta:

```
server:
  error:
     include-message: always
     include-binding-errors: always
     include-stacktrace: on_param
     include-exception: false
```

- La propiedad **include-stacktrace** con valor **on_param**, significa que solo si incluimos el parámetro de seguimiento en la URL (?trace=true), obtendremos la traza completa en la respuesta. El valor **never** indicará que nunca la obtendremos.

La anotación @ExceptionHandler

- La anotación @ExceptionHandler nos permite manejar tipos específicos de excepciones a través de un solo método.

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(MethodArgumentNotValidException.class)
public Map<String, String> handleValidationExceptions(
  MethodArgumentNotValidException ex
) {
    Map<String, String> errors = new HashMap<>();
    ex.getBindingResult().getAllErrors().forEach((error) -> {
        String fieldName = ((FieldError) error).getField();
        String errorMessage = error.getDefaultMessage();
        errors.put(fieldName, errorMessage);
    });
    return errors;
}
```

- Especificamos la excepción **MethodArgumentNotValidException** como la excepción a manejar.
- Spring Boot llamará a este método cuando el objeto especificado no sea válido.

Validando Path Variables y Request Parameters

- En lugar de anotar un campo de clase, agregamos una anotación de restricción (e.g. @Min) directamente al parámetro de método en el controlador Spring:

```
@RestController
@Validated
class ValidateParametersController {

  @GetMapping("/validatePathVariable/{id}")
  ResponseEntity<String> validatePathVariable(
      @PathVariable("id") @Min(5) int id) {
    return ResponseEntity.ok("valid");
  }

  @GetMapping("/validateRequestParameter")
  ResponseEntity<String> validateRequestParameter(
      @RequestParam("param") @Min(5) int param) {
    return ResponseEntity.ok("valid");
  }
}
```

- Tenemos que agregar la anotación **@Validated** de Spring al controlador a nivel de clase para decirle a Spring que evalúe las anotaciones de restricción en los parámetros del método.
- Una validación fallida desencadenará una ConstraintViolationException. Spring no registra un controlador de excepciones predeterminado para esta excepción, por lo que, de forma predeterminada, generará una respuesta con el estado HTTP 500 (Error interno del servidor).

Validando Path Variables y Request Parameters

- Si queremos devolver un estado HTTP 400 (lo que tiene sentido, ya que el cliente proporcionó un parámetro no válido, lo que lo convierte en una solicitud incorrecta), podemos agregar un controlador de excepciones personalizado a nuestro controlador:

```
@RestController
@Validated
class ValidateParametersController {

  // request mapping method omitted

  @ExceptionHandler(ConstraintViolationException.class)
  @ResponseStatus(HttpStatus.BAD_REQUEST)
  ResponseEntity<String> handleConstraintViolationException(ConstraintViolationException e) {
    return new ResponseEntity<>("not valid due to validation error: " + e.getMessage(),
         HttpStatus.BAD_REQUEST);
  }

}
```

Validador personalizado con Spring Boot

- Una anotación de restricción personalizada necesita todo lo siguiente:
  - El parámetro **message**, que apunta a una clave en ValidationMessages.properties, que se utiliza para resolver un mensaje en caso de infracción.
  - El parámetro **groups**, que permite definir en qué circunstancias se activará esta validación.
  - El parámetro **payload**, que permite definir una carga útil para pasar con esta validación (ya que esta es una característica que se usa raramente, no la cubriremos en este tutorial).
  - Una **@Constraint** que apunta a una implementación de la ConstraintValidatorinterfaz.

Validador personalizado con Spring Boot

La anotación de restricción personalizada IpAddress:

```
@Target({ FIELD })
@Retention(RUNTIME)
@Constraint(validatedBy = IpAddressValidator.class)
@Documented
public @interface IpAddress {

  String message() default "{IpAddress.invalid}";

  Class<?>[] groups() default { };

  Class<? extends Payload>[] payload() default { };

}
```

Validador personalizado con Spring Boot

La implementación del validador:

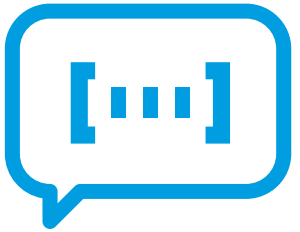
```
class IpAddressValidator implements ConstraintValidator<IpAddress, String> {

  @Override
  public boolean isValid(String value, ConstraintValidatorContext context) {
    Pattern pattern =
      Pattern.compile("^([0-9]{1,3})\\.([0-9]{1,3})\\.([0-9]{1,3})\\.([0-9]{1,3})$");
    Matcher matcher = pattern.matcher(value);
    try {
      if (!matcher.matches()) {
        return false;
      } else {
        for (int i = 1; i <= 4; i++) {
          int octet = Integer.valueOf(matcher.group(i));
          if (octet > 255) {
            return false;
          }
        }
        return true;
      }
    } catch (Exception e) {
      return false;
    }
  }
}
```

Validador personalizado con Spring Boot

- Ahora podemos usar el validador como cualquier otra constraint

```
@IpAddress
private String ipAddress;
```

netmind
a bts company



Internacionalización

08

Internacionalización

- La internacionalización permite que la aplicación se adapte a diferentes idiomas y regiones sin cambios en el código fuente en función de la localización.
- Necesitamos la dependencia Spring Boot Starter Web y Spring Boot Starter Thymeleaf para desarrollar una aplicación web en Spring Boot.

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

<!—Si queremos generar HTML-->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Componentes

- **LocaleResolver:** determina la configuración regional predeterminada de la aplicación.
- **LocaleChangeInterceptor:** se utiliza para cambiar la nueva configuración regional en función del valor del parámetro de idioma agregado a una solicitud.
- **Fuentes de mensajes:** La aplicación Spring Boot de forma predeterminada toma las fuentes de mensajes de la carpeta **src/main/resources** en classpath.
  - El nombre del archivo de mensajes de la configuración regional predeterminada debe ser **message.properties** y los archivos para cada configuración regional deben nombrarse **message_XX.properties** .
  - Todas las propiedades del mensaje deben usarse como valores de pares de claves. Si no se encuentran propiedades en la configuración regional, la aplicación utiliza la propiedad predeterminada del archivo message.properties.

```
welcome.text=Hi Welcome to Everyone
```

```
welcome.text=Salut Bienvenue à tous
```

Componentes

Clase de configuración para la internacionalización

```
@Configuration
public class Internationalization implements WebMvcConfigurer {
   @Bean
   public LocaleResolver localeResolver() {
      SessionLocaleResolver sessionLocaleResolver = new SessionLocaleResolver();
      sessionLocaleResolver.setDefaultLocale(Locale.US);
      return sessionLocaleResolver;
   }
   @Bean
   public LocaleChangeInterceptor localeChangeInterceptor() {
      LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInterceptor();
      localeChangeInterceptor.setParamName("language");
      return localeChangeInterceptor;
   }
   @Override
   public void addInterceptors(InterceptorRegistry registry) {
      registry.addInterceptor(localeChangeInterceptor());
   }
}
```

Componentes

- **Controlador**

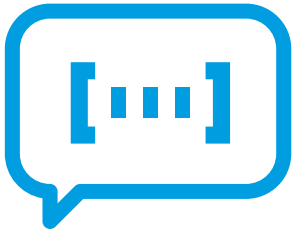
```
@Controller
public class ViewController {
   @RequestMapping("/locale")
   public String locale() {
      return "locale";
   }
}
```

- **Plantilla**

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset = "ISO-8859-1"/>
      <title>Internationalization</title>
   </head>
   <body>
      <h1 th:text = "#{welcome.text}"></h1>
   </body>
</html>
```

netmind
a bts company

09



HTTPS

Habilitar HTTPS

- De manera predeterminada, la aplicación Spring Boot usa el puerto HTTP 8080 cuando se inicia la aplicación.
- Para configurar HTTPS y el puerto 443 en la aplicación Spring Boot:
  - Obtener uncertificado SSL: crear un certificado autofirmado u obtener uno de una autoridad de certificación
  - Habilitar HTTPS y puerto 443

Certificado autofirmado

- Para crear un certificado autofirmado, el entorno Java Run Time viene incluido con la herramienta **keytool** de administración de certificados.

```
keytool -genkey -alias tomcat -storetype PKCS12 -keyalg RSA -keysize 2048 -keystore keystore.p12 -validity 3650
```

  - Este código generará un archivo de almacén de claves PKCS12 llamado **keystore.p12** y el nombre de alias del certificado es tomcat.

Configurar HTTPS

- En el archivo **application.properties** debemos definir el puerto del servidor 443, la ruta del archivo del almacén de claves, la contraseña del almacén de claves, el tipo de almacén de claves y el nombre del alias de la clave.

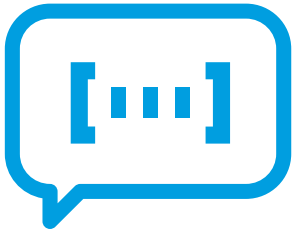
```
server.port: 443
server.ssl.key-store: keystore.p12
server.ssl.key-store-password: springboot
server.ssl.keyStoreType: PKCS12
server.ssl.keyAlias: tomcat
```

- **Archivo yaml**

```
server:
   port: 443
   ssl:
      key-store: keystore.p12
      key-store-password: springboot
      keyStoreType: PKCS12
      keyAlias: tomcat
```

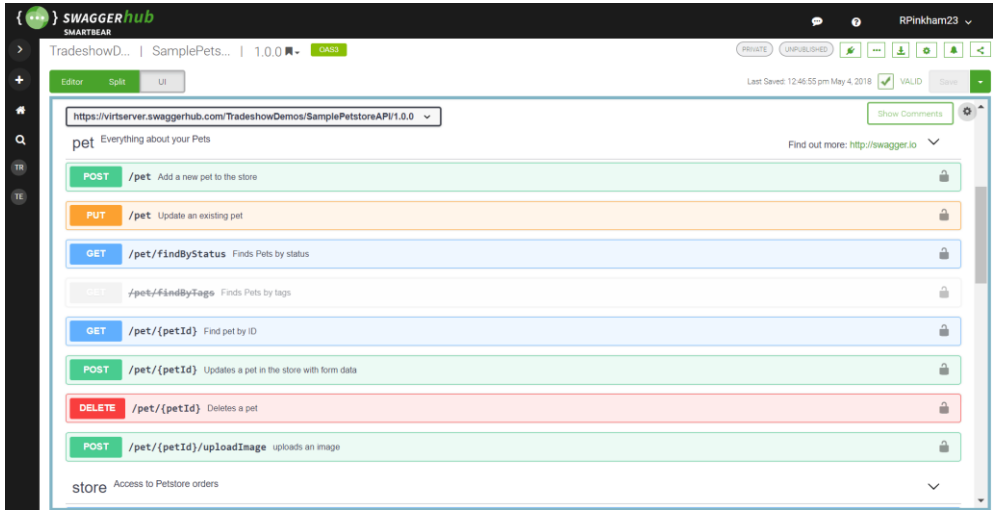
netmind
a bts company

10



Swagger

Swagger



- Swagger2 es un proyecto de código abierto que se utiliza para generar documentos API REST para servicios web RESTful.
- Proporciona una interfaz de usuario para acceder a nuestros servicios web RESTful a través del navegador web.
- https://swagger.io/tools/swagger-ui/

Habilitación de Swagger2

- Para habilitar Swagger2 se debe agregar las siguientes dependencias:

```
<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>
```

- Y agregar la anotación @EnableSwagger2 en la aplicación Spring Boot.

```
@SpringBootApplication
@EnableSwagger2
public class SwaggerDemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(SwaggerDemoApplication.class, args);
   }
}
```

Habilitación de Swagger2

- A continuación, crear el Bean Docket. Necesitamos definir el paquete base para configurar las API REST para Swagger2.

```
@SpringBootApplication
@EnableSwagger2
public class SwaggerDemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(SwaggerDemoApplication.class, args);
   }
   @Bean
   public Docket productApi() {
      return new Docket(DocumentationType.SWAGGER_2)
         .select()
          .apis(RequestHandlerSelectors.any())
          .paths(PathSelectors.any())
          .build();
   }
}
```

  - ⚠ Es posible que sea necesario añadir a las propiedades **application.yml**
    - `spring.mvc.pathmatch.matching-strategy: ANT_PATH_MATCHER`
- Se puede acceder a la ui de la API a través de:
  - http://localhost:8080/swagger-ui.html

Habilitación de Swagger2

En un clase de configuración independiente con API info.

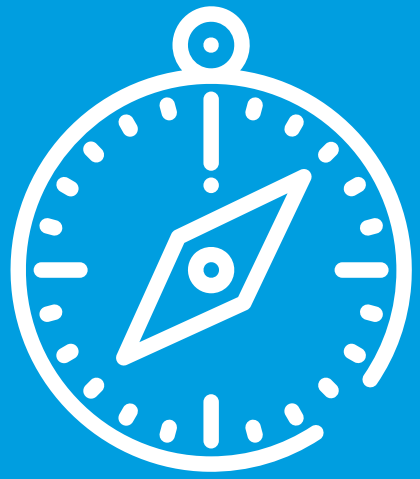
```
@Configuration
@EnableSwagger2
public class SwaggerConfig {

	@Bean
	public Docket apiDocket() {
		return new Docket(DocumentationType.SWAGGER_2)
					.select()
		.apis(RequestHandlerSelectors.basePackage("com.netmind.demo"))
					.paths(PathSelectors.any())
					.build()
					.apiInfo(getApiInfo());
	}

	private ApiInfo getApiInfo() {
		return new ApiInfo(
				"Service API",
				"Service API Description",
				"1.0",
				"http://demo.netmind.com/terms",
				new Contact("Netmind", "https://netmind.com", "apis@netmind.com"),
				"LICENSE",
				"LICENSE URL",
				Collections.emptyList()
		);
	}
}
```

Agregar descripción a métodos y parámetros

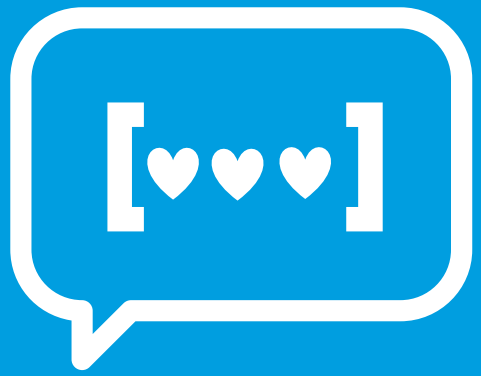
```
@ApiOperation(value = "Get a product by id", notes = "Returns a product as per the id")
@ApiResponses(value = {
  @ApiResponse(code = 200, message = "Successfully retrieved"),
  @ApiResponse(code = 404, message = "Not found - The product was not found")
})
@GetMapping(value = "/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity getProduct(@PathVariable @Min(1) @ApiParam(name = "id", value = "Product id", example = "1")
Long id) {
...
}
```

Agregar descripción y ejemplos al modelo

```
@ApiModelProperty(notes = "Product ID", example = "1", required = true)
private Long id;

@ApiModelProperty(notes = "Product name", example = "Product 1", required = false)
private String name;
```



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

netmind
a bts company

Thanks!

Follow us: