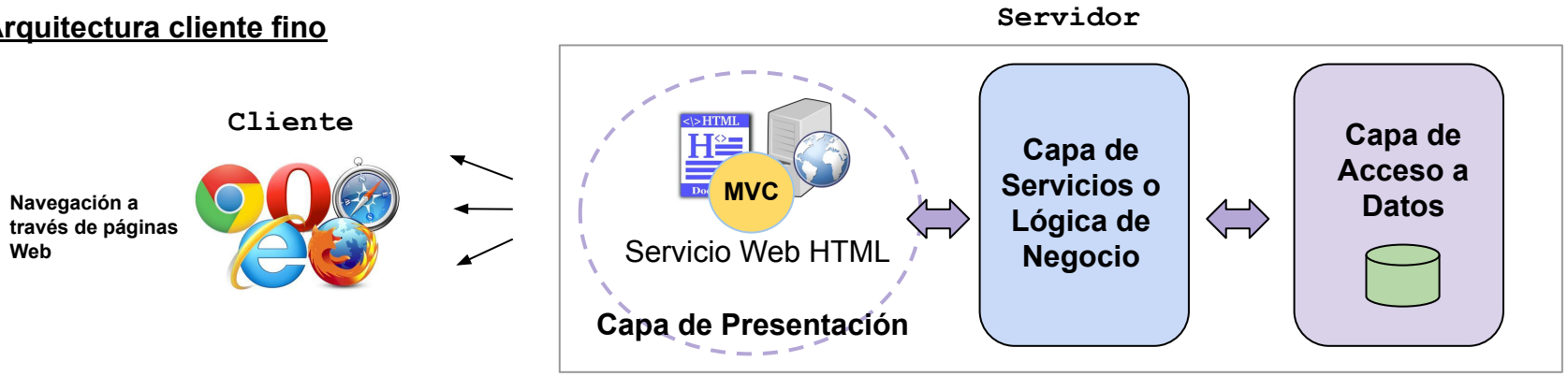
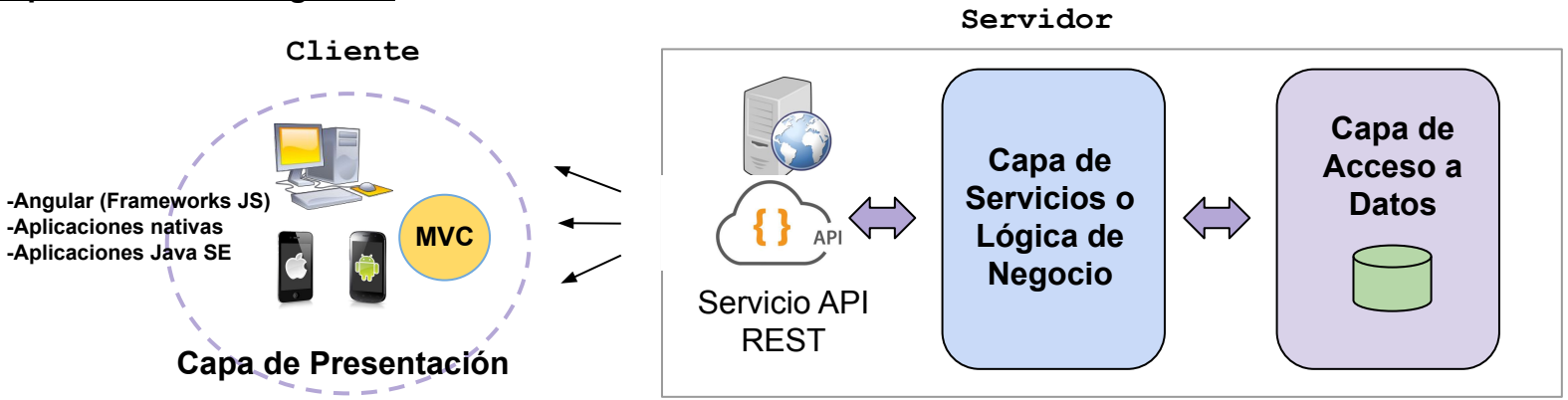
Spring MVC y REST

Arquitecturas de aplicaciones Web

Arquitectura cliente fino



Arquitectura cliente grueso



Spring MVC

¿Qué es Spring MVC?

Es un framework de aplicaciones Web basado en el patrón Model-View-Controller y en la atención de requerimientos.

¿Qué nos provee Spring MVC?

Nos permite construir la capa de presentación de aplicaciones basadas en la **arquitectura cliente fino** como así también servicios RESTFul para las basadas en la **arquitectura cliente grueso**.

ReST

Representational State Transfer

¿Qué es REST?

REST es un estilo arquitectural para diseñar Web Services.

REST re-significó el protocolo HTTP.

Los lineamientos arquitecturales que sigue REST, son:

- uso de recursos direccionables
- uso de una interfaz uniforme y acotada
- es orientado a representación
- uso de un protocolo de comunicación sin estado

¿Qué es un Servicio Web?

El World Wide Web Consortium (W3C) define un servicio web como un sistema de software designado para dar soporte a la interacción de máquina a máquina interoperativa a través de una red.

Un web service es un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos

La propuesta de RESTful redefine los “servicios web” para trabajar mejor en la WEB, promoviendo la escalabilidad, la performance y la modificabilidad.

Generalidades del estilo REST

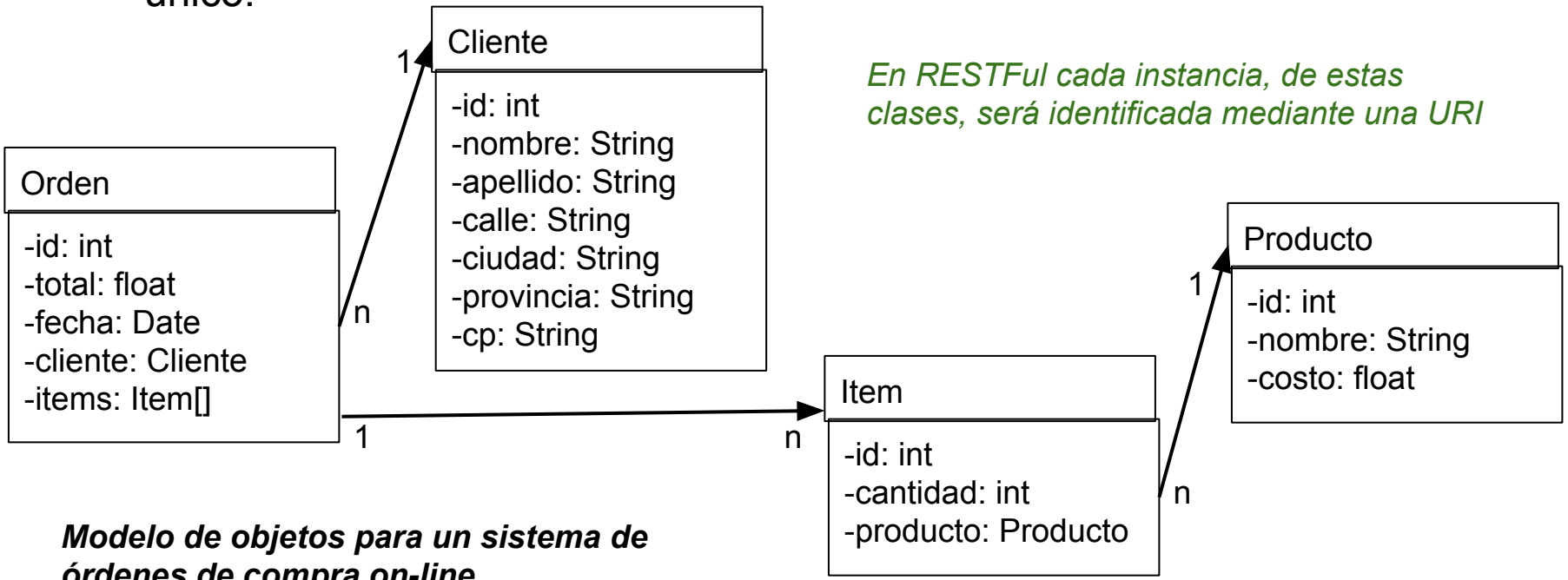
Las aplicaciones RESTFul son **cliente-servidor**.

La comunicación entre el **cliente** y el **servidor** es a través de un **protocolo sin estado**, típicamente HTTP.

El **cliente** y el **servidor** intercambian recursos mediante una interface estandarizada.

Principios de REST

1) **Identificación de recursos mediante URIs**: en REST cada objeto o recurso es alcanzable mediante el uso de una URI como identificador único.



En RESTFul cada instancia, de estas clases, será identificada mediante una URI

Modelo de objetos para un sistema de órdenes de compra on-line

Identificación de recursos mediante URIs

http://rest.com/ordenes/233

```
{
  "id": "233",
  "url": "http://rest.com/ordenes/233",
  "total": "$199.02",
  "fecha": "22/12/2015 06:56",
  "cliente": {
    "id": "117",
    "url": "http://rest.com/clientes/117",
    "nombre": "Diego",
    "apellido": "Capusoto",
    "calle": "Corrientes",
    "ciudad": "Buenos Aires",
    "provincia": "Buenos Aires",
    "cp": "1043"
  },
  "items": {
    "item": {
      "id": "144",
      "producto": {
        "id": "543",
        "url": "http://rest.com/productos/543",
        "nombre": "iPhone",
        "costo": "$199.99"
      }
    }
  ...
  }
}
```

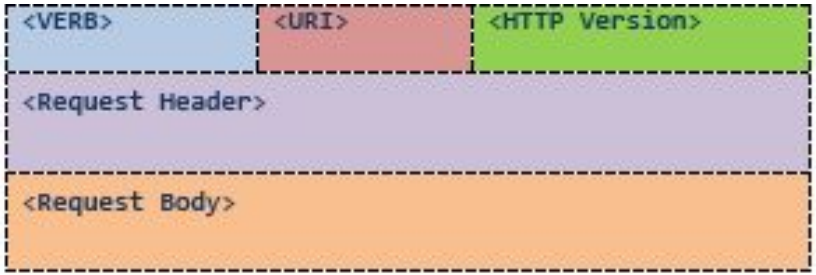
```
{
  "id": "117",
  "url": "http://rest.com/clientes/117",
  "nombre": "Diego",
  "apellido": "Capusoto",
  "calle": "Corrientes",
  "ciudad": "Buenos Aires",
  "provincia": "Buenos Aires",
  "cp": "1043"
}
```

```
{
 "id": "543",
 "url": "http://rest.com/productos/543",
 "nombre": "iPhone",
 "costo": "$199.99"
}
```

URIs RESTFul que identifican los objetos del modelo

Principios de REST

2) **Uso de una interfaz uniforme y acotada**: los recursos son manipulados usando un conjunto fijo de cuatro operaciones que permiten su creación, lectura, actualización y eliminación.
RESTFul implementa esta funcionalidad mediante los métodos **`PUT`**, **`GET`**, **`POST`** y **`DELETE`** de HTTP.



<VERB>: GET, PUT, POST, DELETE, HEAD y OPTIONS
<Request Body>: se especifican los recursos a transferir
<URI> es la URI del recurso sobre el cual se realiza la operación

Principios de REST

GET

Se utiliza para consultar por información específica
Es una operación de sólo lectura
Es idempotente y segura

POST

Se utiliza para almacenar una nueva entidad de datos como un nuevo recurso que se identificara con una URI generada por el servidor
Es no-idempotente e inseguro

Principios de REST

PUT

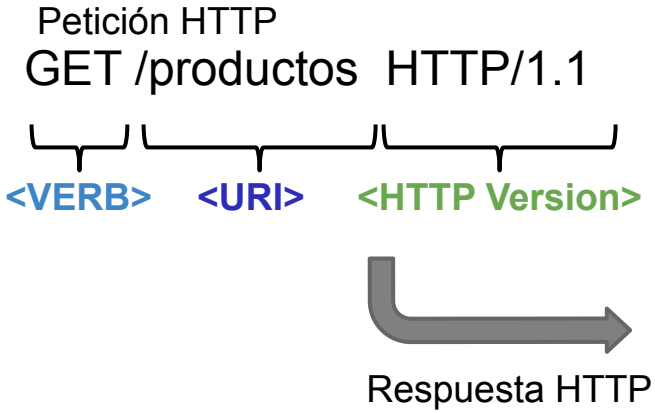
Se utiliza para almacenar el cuerpo del mensaje bajo la ubicación provista en el mensaje HTTP.
Se utiliza para *updates*
Es idempotente, porque ejecutarlo más de una vez no afecta al servicio

DELETE

Se utiliza para eliminar recursos identificados por una URI
Es idempotente

Uso de una interfaz uniforme y acotada

¿Cómo consultar todos los productos disponibles en el sistema?

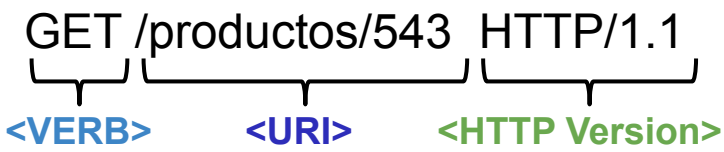



```
HTTP/1.1 200 OK
Content-Type: application/json

[
   {
      "id": "543",
      "url": "http://rest.com/productos/543",
      "nombre": "iPhone",
      "costo": "$199.99"
   },
   {
      "id": "544",
      "url": "http://rest.com/productos/544",
      "nombre": "Samsung",
      "costo": "$159.99"
   }
   ...
]
```

Uso de una interfaz uniforme y acotada

¿Cómo consultar un producto particular?

Petición HTTP



Respuesta HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json

{
 "id": "543",
 "url": "http://rest.com/productos/543",
 "nombre": "iPhone",
 "costo": "$199.99"
}
```

Uso de una interfaz uniforme y acotada

¿Cómo crear un producto como un nuevo recurso?

POST /productos/ HTTP/1.1

<VERB> <URI> <HTTP Version>

La respuesta HTTP incluye la nueva URI del recurso creado, para que el cliente pueda referenciarlo

```
POST /productos/ HTTP/1.1
Content-Type: application/json

{
 "nombre": "Alcatel",
 "costo": "$100.99"
}
```

Petición HTTP

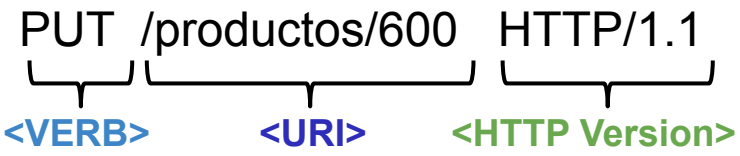
```
HTTP/1.1   201  Created
Content-Type: application/json
Location: http://rest.com/productos/600

{
 "id": "600",
 "url": "http://rest.com/productos/600",
 "nombre": "Alcatel",
 "costo": "$100.99"
}
```

Obligatorio

Respuesta HTTP

Uso de una interfaz uniforme y acotada

¿Cómo actualizar los datos de un producto existente?




```
PUT /productos/600 HTTP/1.1
Content-Type: application/json

{
 "id": "600",
 "url": "http://rest.com/productos/600",
 "nombre": "Alcatel",
 "costo": "$123.99"
}
```

Petición HTTP

```
HTTP/1.1   204  No Content
```

ó

```
HTTP/1.1   200  OK
Content-Type: application/json

{
 "id": "600",
 "url": "http://rest.com/productos/600",
 "nombre": "Alcatel",
 "costo": "$123.99"
}
```

Dos posibles respuestas HTTP

Uso de una interfaz uniforme y acotada

¿Cómo eliminar un producto existente?

Petición HTTP

DELETE /productos/545 HTTP/1.1

<VERB> <URI> <HTTP Version>

Dos posibles respuestas HTTP

```
HTTP/1.1   204  No Content
```

ó

```
HTTP/1.1   200  OK
Content-Type: application/json

{
 "id": "545",
 "url": "http://rest.com/productos/545",
 "nombre": "Nokia",
 "costo": "$137.99"
}
```

Interfaz uniforme y acotada

Las **operaciones** de nuestro *sistema de órdenes de compra on-line* se ajustan correctamente a los métodos del protocolo HTTP: se utiliza **GET** para leer, **PUT** para actualizar, **POST** para crear y **DELETE** para remover.

Si bien es posible utilizar parámetros en las peticiones HTTP, REST propone no cambiar la semántica de los métodos HTTP. Por ejemplo agregar un parámetro de nombre “action” al GET indicando la acción de actualizar un recurso, no cumple los principios de REST. En este caso se debería usar el método PUT.

Un buen uso de parámetros en los métodos HTTP, por ejemplo es recuperar todos los productos cuyo nombre comienza con A, B, C, o D

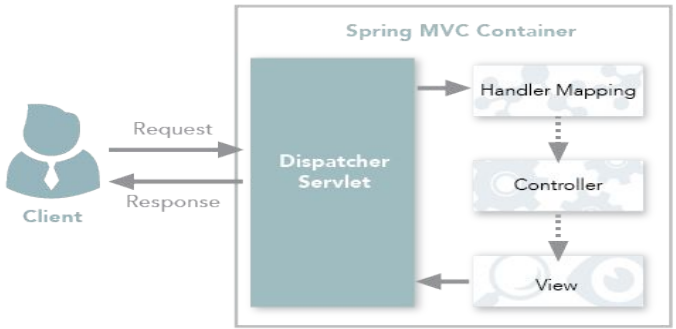
GET /productos?start=A&end=D HTTP/1.1

Implementando REST con Spring MVC

El Framework Web de Spring
Spring MVC

El framework web de Spring o Spring MVC, está basado en el patrón Model-View-Controller (MVC) y ayuda a construir aplicaciones web flexibles y poco acopladas.
El patrón MVC separa diferentes aspectos de la aplicación y promueve un acoplamiento débil entre ellos.

- El **modelo** (*model*) encapsula los datos de la aplicación que en general serán POJOs.
- La **vista** (*view*) es responsable de mostrar los datos del modelo y en general genera HTML que el navegador del cliente puede interpretar.
- El **controlador** (*controller*) que es el responsable de procesar los requerimientos del usuario y de construir un modelo apropiado y pasarlo a la vista para que lo muestre.



La imagen muestra a muy alto nivel el procesamiento de un requerimiento en Spring MVC.
El framework Spring MVC implementa el patrón de diseño “Front Controller” y el rol central lo cumple el DispatcherServlet (este patrón es implementado por muchos framework web).
El DispatcherServlet está completamente integrado con el Contenedor IoC de Spring y por esto utiliza todas las características que brinda Spring.

Spring MVC
Contextos de la aplicación

Para cualquier aplicación de Spring que usa Spring MVC comúnmente se definen dos contextos: ApplicationContext CORE y ApplicationContext para MVC. El listener inspecciona el parámetro de inicialización **contextConfigLocation**. Si el parámetro no existe, el listener usa **/WEB-INF/applicationContext.xml** como valor por defecto.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
  <context-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>/WEB-INF/spring/applicationContext.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

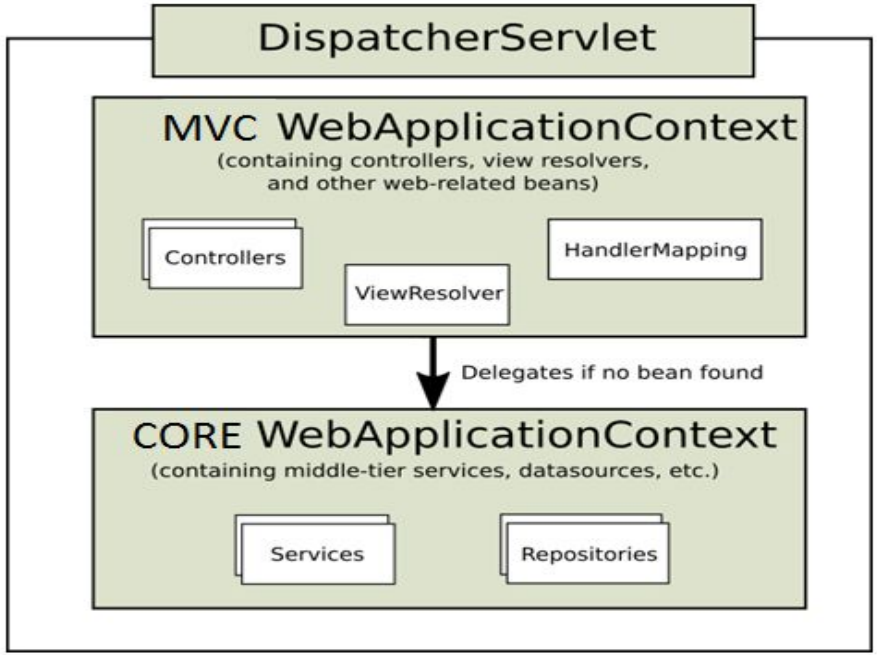
Configuracíón del ApplicationContext primario. El listener carga los beans DAOs, Servicios, etc.

Configuración del MVC ApplicationContext. Se cargan controllers, view resolvers, handler mappings, etc.

Spring MVC
Contextos de la aplicación

En Spring MVC, cada **`DispatcherServlet`** tiene su propio **`ApplicationContext`**, el cual hereda todos los beans definidos en el **`ApplicationContext`** primario. Lo más común es tener un solo **`DispatcherServlet`**.

Jerarquía de contextos típica en Spring MVC



El ApplicationContext para MVC es cargado via el DispatcherServlet de Spring.

El contexto primario o CORE de Spring, es cargado via el ContextLoaderListener de Spring.

Spring MVC
Contextos de la aplicación
Configuración con Anotaciones

Spring

Configuración del Contexto con Anotaciones (1/2)

Ahora crearemos una configuración basada en java. Para ello se crea una clase **AppConfig**, donde se registrarán todos los bean relacionados con Spring usando un estilo de configuración basado en JAVA.

```java
package ttps.spring.config;

import java.util.List;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "ttps.spring")
public class AppConfig implements WebMvcConfigurer {

    @Override
    public void extendMessageConverters(List<HttpMessageConverter<?>> converters) {
        converters.add(new MappingJackson2HttpMessageConverter());
    }
}
```

Tag XML	Anotación	Descripción
<context:component-scan/>	@ComponentScan	Escanea por controllers, repositorios, servicios, beans, etc.
<mvc:annotation-driven/>	@EnableWebMvc	Habilita anotaciones como @Controller de SpringMVC
spring_mvc-xml	@Configuration	Trata a la clase como un archivo de configuracion para app String MVC

Spring

Configuración del Contexto con Anotaciones (2/2)

Ahora crearemos una clase JAVA para reemplazar a nuestro **`web.xml`**. Esta clase debe implementar la interface **`org.springframework.web.WebApplicationInitializer`**.

```java
package ttps.spring.config;

import javax.servlet.ServletContext;

public class SpringWebApp implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) throws ServletException {
        // Create the 'root' Spring application context
        AnnotationConfigWebApplicationContext rootContext = new AnnotationConfigWebApplicationContext();
        rootContext.register(AppConfig.class);

        //ContextLoaderListener - Manage the lifecycle of the root application context
        container.addListener(new ContextLoaderListener(rootContext));

        //DispatcherServlet - Register and map the dispatcher servlet
        ServletRegistration.Dynamic dispatcher = container.addServlet("DispatcherServlet", new DispatcherServlet(rootContext));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }

}
```

Esta solución de Spring está basada en Servlet 3.0

```java
public interface ServletContainerInitializer {

public void onStartup(Set<Class<?>> c, ServletContext ctx)
      throws ServletException;
}
```

Este método es invocado por Tomcat al iniciar la aplicación. Permite registrar servlets, filtros y listeners en tiempo de ejecución evitando la necesidad del archivo web.xml o anotaciones en las clases.

En esta clase se registra la clase **AppConfig** como la que configura el contexto y registramos y configuramos el DispatcherServlet, el cual actúa como **FrontController** para las aplicaciones Spring MVC.

Spring MVC
Controllers

Spring MVC *delega la responsabilidad de manejar los requerimientos a los Controllers*. Los controllers son parecidos a los servlets, se mapean a una o más URIs, fueron construidos para abstraer objetos **`HTTPServletRequest`** y **`HTTPServletResponse`**, son *singletons* y manejan múltiples requerimientos. La API de los controllers no intenta ocultar su dependencia de servlets sino que las toma y las potencia.

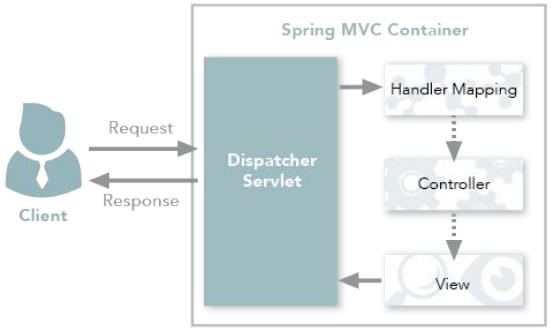
Los Controllers son los responsables de procesar los requerimientos HTTP, hacer el trabajo necesario, crear objetos para la respuesta, y pasar el control de vuelta al Front Controller.

- Los Controllers no manejan las vistas, se centran en el manejo de peticiones y delegan en la capa de servicio.
- Los Controllers se implementan usando anotaciones como **`@Controller`**, **`@RequestMapping`**, **`@RequestParam`**, **`@ModelAttribute`**, etc. Los controladores implementados de esta forma no tienen que extender de ninguna clase ni implementar interfaces específicas.

Implementando REST con Spring MVC

La principal diferencia entre el tradicional Spring MVC controller y el RESTful web service controller es la forma en que se crea el body de la respuesta HTTP.

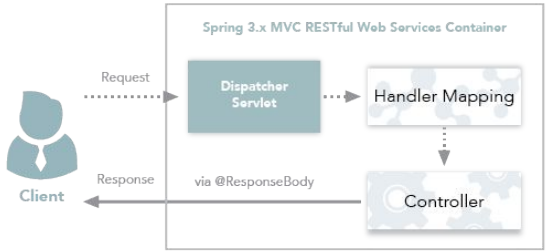
Spring MVC controller tradicional



Los requerimientos son procesados por el **Controller** y la respuesta es retornada al **DispatcherServlet** el cual **devuelve la vista**.

En RESTful, el controller simplemente retorna el objeto, y los datos del objeto son escritos directamente en la respuesta HTTP en formato JSON/XML.

Usando la anotación @ResponseBody

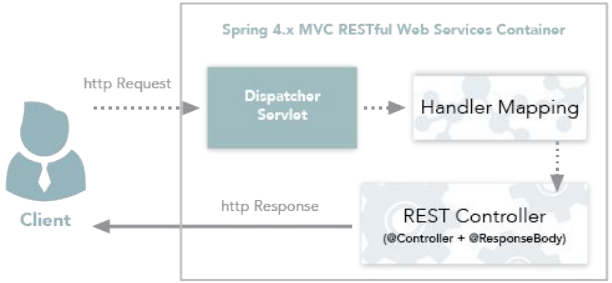


Usando la anotación **@ResponseBody** se indica que **el objeto retornado** debe ser escrito **directamente en el body de la respuesta HTTP**.

Cada método en el Controller debe ser anotado con @ResponseBody para obtener este efecto.

El método preferido para crear RESTful web services es usar @RestController.

Usando la anotación @RestController



Al anotar la clase controller con **@RestController** ya no se necesita anotar con @ResponseBody todos los métodos. **@ResponseBody esta activo por default**.

@RestController está disponible desde Spring v4.0.

Implementando REST con Spring MVC

@RequestBody: se utiliza en parámetros de métodos para indicar que el valor del parámetro debe obtenerse del body del requerimiento HTTP. Spring convierte el body del requerimiento HTTP en un objeto.

```
@RequestMapping(value = "/users/{id}", method = RequestMethod.PUT)
    public ResponseEntity<User> updateUser(@PathVariable("id") long id, @RequestBody User user) {
        User currentUser = userService.findById(id);
         . . .
        userService.updateUser(currentUser);
        return new ResponseEntity<User>(currentUser, HttpStatus.OK);
}
```

..con user se actualizará el objeto currentUser

@ResponseBody: es similar a @RequestBody, es una anotación de método que sirve para indicar que el objeto que retorna el método debe ser escrito directamente en el body de la respuesta HTTP no en el modelo, ni que se interprete como nombre de vista). Con @RestController ya no se necesita anotar con @ResponseBody todos los métodos, **está activo por default**.

Implementando REST con Spring MVC

@RestController: es una anotación de Spring 4 que elimina la necesidad de anotar cada método con @ResponseBody, puede ser considerada una combinación de @Controller y @ResponseBody.

HttpMessageConverter: es una interface que permite la conversión de distintos tipos de medios a objetos, provenientes de los requerimientos HTTP y hacia las respuestas HTTP. Algunas implementaciones son:

StringHttpMessageConverter: todos los tipos de medios de texto, contenidos de tipo *text/**

FormHttpMessageConverter: para contenido *application/x-www-form-urlencoded*

ByteArrayHttpMessageConverter: para contenido *application/octet-stream*

MarshallingHttpMessageConverter: para contenido *text/xml, application/xml*

MappingJackson2HttpMessageConverter: para contenido *application/json*

BufferedImageHttpMessageConverter: puede escribir y leer *java.awt.image.BufferedImage* desde los requerimientos/respuestas

Implementando REST con Spring MVC

HttpEntity: representa un requerimiento o respuesta HTTP permitiendo acceder a los headers y body del requerimiento y respuesta HTTP. **Tiene como subclases a RequestEntity**, **ResponseEntity**.

ResponseEntity: representa la respuesta entera HTTP, a diferencia de HttpEntity **permite manipular el código de respuesta, los headers y el body de la respuesta HTTP**.

```
@PostMapping
 public ResponseEntity<void> createUser(@RequestBody User user) {

    ….
    return new ResponseEntity<User>(headers, HttpStatus.CREATED);
 }
```

Códigos de Respuesta HTTP

1xx: Respuestas informativas

2xx: Peticiones correctas

3xx: Redirecciones

4xx: Errores del cliente

5xx: Errores de servidor

*org.springframework.http.***HttpStatus** permite representar estos estados, por ejemplo:

HttpStatus.OK ⇒ `200 OK`

HttpStatus.CREATED ⇒ `201 Created`

HttpStatus.NO_CONTENT ⇒ `204 No Content`

HttpStatus.NOT_FOUND ⇒ `404 Not Found`

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/HttpStatus.html

Ejemplo de REST con Spring MVC

Ejemplo de un REST que permite hacer lo siguiente:

GET con URI: **/users/** retorna una lista de usuarios

GET con URI: **/users/1** retorna el usuario con ID 1

POST con URI: **/users/** con un objeto usuario en formato JSON crea un nuevo usuario

PUT con URI: **/users/3** con un objeto usuario en formato JSON actualiza el usuario con ID 3

DELETE con URI: **/users/4** elimina el usuario con ID 4

DELETE con URI: **/users/** elimina todos los usuarios

Ejemplo de REST con Spring MVC

```
package com.ejemplo.restful.springmvc;
import org.springframework.web.bind.annotation.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import java.util.List;

@RestController
@RequestMapping("/users", produces =
                    MediaType.APPLICATION_JSON_VALUE)
public class UsuarioRestController {

 @Autowired
  UserService userService;

 //Recupero todos los usuarios
@GetMapping
public ResponseEntity<List<User>> listAllUsers() {
   List<User> users = userService.findAllUsers();
   if(users.isEmpty()){
        return  new
             ResponseEntity<List<User>>(HttpStatus.NO_CONTENT);
    }
     return new ResponseEntity<List<User>>(users, HttpStatus.OK);
  }
```

representa la respuesta entera HTTP permite manipular el código de respuesta, los headers y el body de la respuesta HTTP

```
 //Recupero un usuario dado
@GetMapping("/{id}")
 public ResponseEntity<User> getUser(@PathVariable("id") long id) {

   System.out.println("Obteniendo usuario con id " + id);
   User user = userService.findById(id);
   if (user == null) {
         System.out.println("Usuario con id " + id + " no encontrado");
         return new ResponseEntity<User>(HttpStatus.NOT_FOUND);
    }

    return new ResponseEntity<User>(user, HttpStatus.OK);
}

. . .
```

@RequestParam: se utiliza para obtener parámetros de una URL.
Por ej. para http://www.ej.com/alumnos?legajo=123&... **@RequestParam**("legajo")
Para mas información ver documentacion de la API:
http://docs.spring.io/spring/docs/current/javadoc-api/

Ejemplo de REST con Spring MVC

```
//Creo un usuario

@PostMapping
 public ResponseEntity<void> createUser(@RequestBody User user) {

    System.out.println("Creando el usuario"  + user.getName());

    if (userService.isUserExist(user)) {
        System.out.println("Ya existe un usuario con nombre " + user.getName());
        return new ResponseEntity<void>(HttpStatus.CONFLICT);      //Código de respuesta 409
    }

    userService.saveUser(user);
    return new ResponseEntity<User>(headers, HttpStatus.CREATED);
 }
```

Ejemplo de REST con Spring MVC

```
//Actualizo un usuario

@PutMapping("/{id}")
public ResponseEntity<User> updateUser(@PathVariable("id") long id, @RequestBody User user) {

    System.out.println("Actualizando el usuario " + id);

    User currentUser = userService.findById(id);

    if (currentUser==null) {
        System.out.println("User with id " + id + " not found");
        return new ResponseEntity<User>(HttpStatus.NOT_FOUND);
    }

    currentUser.setName(user.getName());
    currentUser.setAge(user.getAge());
    currentUser.setSalary(user.getSalary());

    userService.updateUser(currentUser);
    return new ResponseEntity<User>(currentUser, HttpStatus.OK);
}
```

Ejemplo de REST con Spring MVC

```
//Elimino un usuario

@DeleteMapping("/{id}")
public ResponseEntity<User> deleteUser(@PathVariable("id") long id) {

    System.out.println("Obteniendo y eliminando el usuario con id " + id);

    User user = userService.findById(id);
    if (user == null) {
        System.out.println("No es posible eliminar, no se encuentra el usuario  con id " + id);
        return new ResponseEntity<User>(HttpStatus.NOT_FOUND);
    }

    userService.deleteUserById(id);

    return new ResponseEntity<User>(HttpStatus.NO_CONTENT);
}
```

Ejemplo de REST con Spring MVC

```
    //Elimino todos los usuarios

    @DeleteMapping
    public ResponseEntity<User> deleteAllUsers() {
        System.out.println("Eliminando todos los usuarios");

        userService.deleteAllUsers();

        return new ResponseEntity<User>(HttpStatus.NO_CONTENT);
    }

} // fin de la clase
```

Para trabajar con JSON debemos contar con las librerías:

- **jackson-databind**
- **jackson-core**
- **jackson-annotations**

Librería Jackson

La librería Jackson provee conversión automática de JSON a clases Java y viceversa de manera directa. https://github.com/FasterXML/jackson.

Con las anotaciones de Jackson podemos controlar la serialización y deserialización entre clases POJO y JSON.

Tener en cuenta las anotaciones para evitar bucles:

@JsonIgnoreProperties anotación a nivel de clase, para indicar una lista de atributos que no deben serializarse

@JsonIgnore anotación a nivel atributo, para indicar que el atributo no debe serializarse

```
@JsonIgnoreProperties({ "id" })
public class BeanWithIgnore {

    public int id;

    public String name;
}
```

```
public class BeanWithIgnore {
    @JsonIgnore
    public int id;

    public String name;
}
```

Spring MVC - @RequestHeader

Con Spring MVC es posible utilizar la anotación **@RequestHeader** para acceder de manera directa a los atributos del encabezado HTTP.

```
@GetMapping
public ResponseEntity<List<User>> listAllUsers(@RequestHeader("token")  String token,
                                @RequestHeader Map<String, String> mapHeaders) {

        System.out.println("Token: " + token);

        //acceso a todos los atributos del encabezado HTTP
        for (Map.Entry<String, String> entry : mapHeaders.entrySet()) {
                System.out.println(entry.getKey() + "  : " + entry.getValue());
        }

        ……
}
```

Spring MVC - Manejo de Excepciones

Con Spring MVC es posible enviar una respuesta HTTP levantando una excepción de tipo **ResponseStatusException**. Es una manera flexible y directa de enviar una respuesta con un código de error adecuado.
HandlerExceptionResolver intercepta y procesa cualquier excepción levantada y no manejada por un controller

```
@RestController
@RequestMapping("/api/usuarios")
public class UsuarioController {

    @Autowired
    private UsuarioService usuarioService;


@GetMapping("/{id}")
public ResponseEntity<Usuario> getUsuarioById(@PathVariable("id") Long id) {
      try {
         return new ResponseEntity<>(usuarioService.getUsuarioById(id),HttpStatus.OK);
      } catch (Exception e) {
         throw new ResponseStatusException( HttpStatus.NOT_FOUND, "Usuario no encontrado");
      }
}
```

Spring MVC - Manejo de Excepciones

Con Spring MVC es posible manejar las excepciones de manera **centralizada** utilizando: **`@ControllerAdvice`** (ó **`@RestControllerAdvice`**) y **`@ExceptionHandler`**.

`@ControllerAdvice` es una anotación en Spring que permite manejar excepciones de forma global para todos los controladores. Se puede usar con controladores MVC tradicionales o controladores REST. Si se utiliza controladores @RestController es equivalente a `@RestControllerAdvice`

`@RestControllerAdvice` es una especialización de @ControllerAdvice que está diseñada específicamente para controladores REST. Siempre convierte las respuestas de error a JSON o al formato que el cliente solicita en la cabecera Accept

```
@RestControllerAdvice
public class GlobalExceptionHandler {
    // Métodos de manejo de excepciones
}
```

Spring MVC - Manejo de Excepciones

@ExceptionHandler es una anotación que se utiliza dentro de **@ControllerAdvice** o en un controlador específico para manejar excepciones particulares.

Se especifica el tipo de excepción que manejará cada método.

```
@ExceptionHandler(EntityNotFoundException.class)
public ResponseEntity<String> handleEntityNotFound(EntityNotFoundException ex) {
    return new ResponseEntity<>("Entidad no encontrada", HttpStatus.NOT_FOUND);
}
```

Spring MVC - Manejo de Excepciones

Ejemplo de @ExceptionHandler dentro de un controlador. Cuando se envía un POST con datos de entrada que deben cumplir ciertas validaciones

```
@RestController
@RequestMapping("/api/usuarios")
public class UsuarioController {

  @PostMapping
  public ResponseEntity<Usuario> crearUsuario(@Valid @RequestBody Usuario usuario) {
    // Lógica de negocio
  }

  @ExceptionHandler(MethodArgumentNotValidException.class)
  public ResponseEntity<String> handleValidationExceptions(MethodArgumentNotValidException ex) {
      return new ResponseEntity<>("Datos de entrada inválidos", HttpStatus.BAD_REQUEST);
  }
..
```

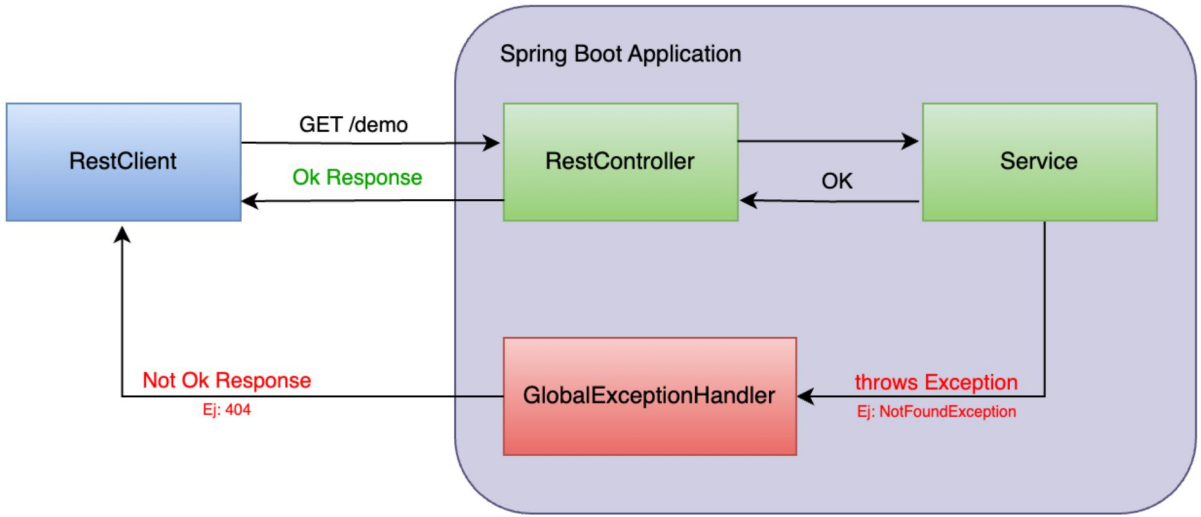
//Ejemplo de salida de error para validación

```
HTTP/1.1   400  Bad Request
Content-Type: application/json

{
    "status": 400,
    "error": "Bad Request",
    "message": "Datos de entrada inválidos"
}
```

Spring MVC - Manejo de Excepciones

Ejemplo de @ExceptionHandler dentro de una clase anotada con `@RestControllerAdvice`

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>> handleUsuarioYaExiste(UsuarioYaExisteException ex) {

...
```



Spring MVC - Manejo de Excepciones

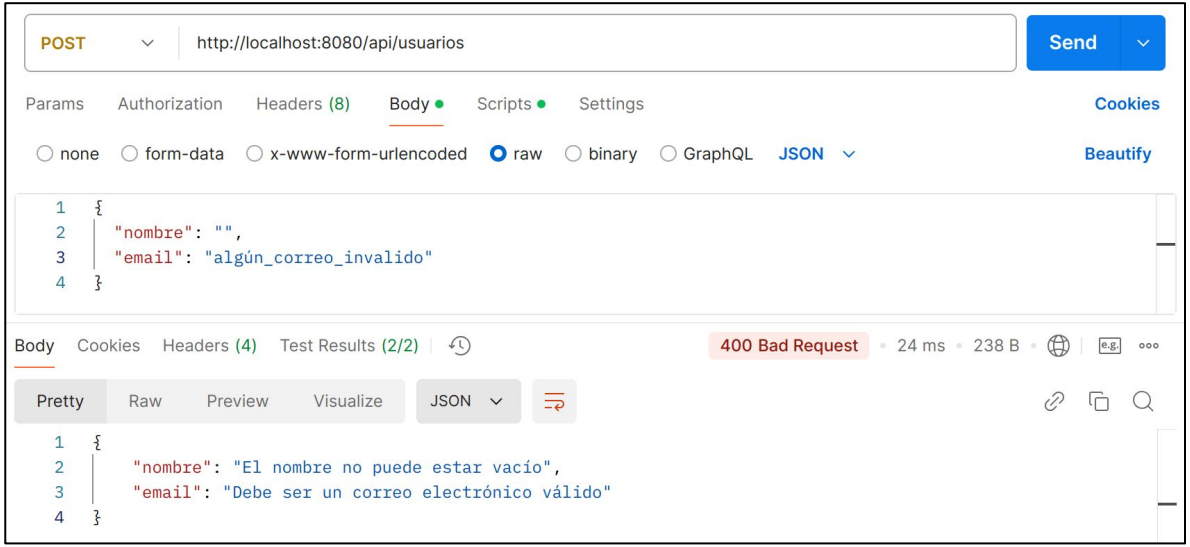
Ejemplo de manejo de excepciones centralizado, provenientes de peticiones a REST

```java
@RestController
@RequestMapping("/api/usuarios")
public class UsuarioController {

 @Autowired
 private UsuarioService usuarioService;

 @PostMapping
 public ResponseEntity<Usuario> crearUsuario(@Valid @RequestBody Usuario usuario) {
     return new ResponseEntity<>(usuarioService.crearUsuario(usuario), HttpStatus.CREATED);
 }
```

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

 // Manejo de excepciones de validación de @Valid
 @ExceptionHandler(MethodArgumentNotValidException.class)
 public ResponseEntity<Map<String, String>> handleValidationExceptions(MethodArgumentNotValidException ex) {
     Map<String, String> errores = new HashMap<>();
     ex.getBindingResult().getAllErrors().forEach((error) -> {
         String campo = ((FieldError) error).getField();
         String mensaje = error.getDefaultMessage();
         errores.put(campo, mensaje);
     });
     return new ResponseEntity<>(errores, HttpStatus.BAD_REQUEST);
 }
```

Spring MVC - Manejo de Excepciones

```java
import jakarta.persistence.*;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

@Entity
public class Usuario {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 @NotBlank(message = "El nombre no puede estar vacío")
 @Size(min = 2, message = "El nombre debe tener al menos 2 caracteres")
 private String nombre;

 @NotBlank(message = "El correo no puede estar vacío")
 @Email(message = "Debe ser un correo electrónico válido")
 @Column(unique = true)
 private String email;
```



Spring MVC - Manejo de Excepciones

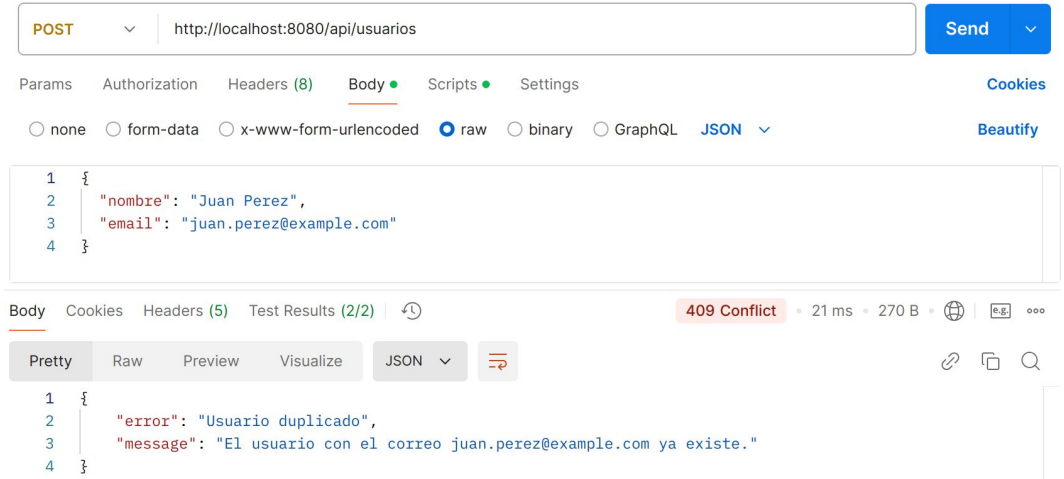
Ejemplo de manejo de excepciones centralizado, provenientes de la capa de servicio o DAO

```java
@Service
public class UsuarioService {

 @Autowired
 private UsuarioRepository usuarioRepository;

 public Usuario crearUsuario(Usuario usuario) {
     if (usuarioRepository.existsByEmail(usuario.getEmail())) {
         throw new UsuarioYaExisteException("El usuario con el correo " + usuario.getEmail() + " ya existe.");
     }
     return usuarioRepository.save(usuario);
 }
```

```java
@SuppressWarnings("serial")
public class UsuarioYaExisteException extends RuntimeException {
 public UsuarioYaExisteException(String mensaje) {
     super(mensaje);
 }
}
```

Spring MVC - Manejo de Excepciones

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

 // Manejo de la excepción personalizada de usuario duplicado
 @ExceptionHandler(UsuarioYaExisteException.class)
 public ResponseEntity<Map<String, String>> handleUsuarioYaExiste(UsuarioYaExisteException ex) {
     Map<String, String> error = new HashMap<>();
     error.put("error", "Usuario duplicado");
     error.put("message", ex.getMessage());
     return new ResponseEntity<>(error, HttpStatus.CONFLICT);
 }
```



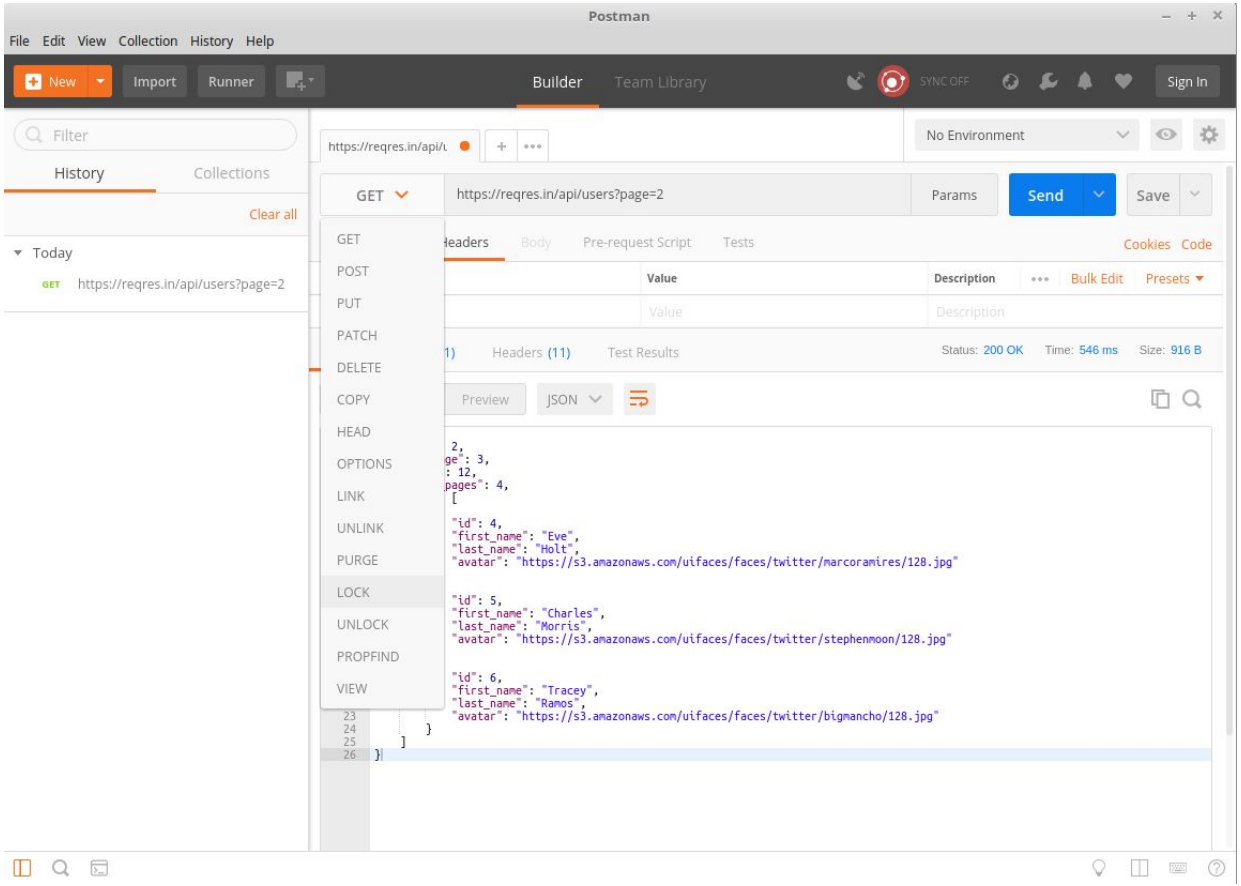
Spring MVC - Manejo de Excepciones

Algunas de las excepciones típicas que podrían manejarse con @ExceptionHandler

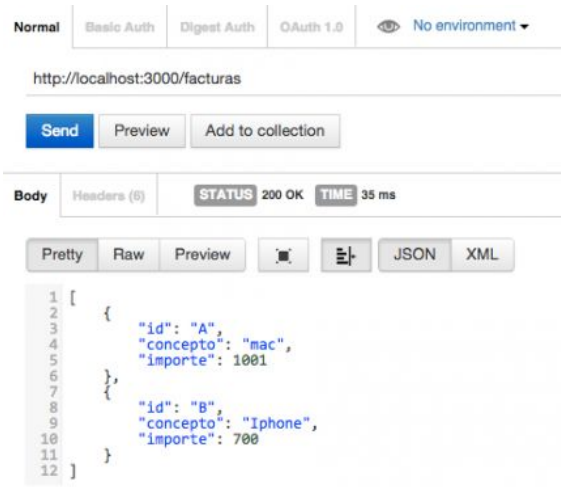
- `HttpRequestMethodNotSupportedException`: cuando un método HTTP no es compatible con la operación solicitada (por ejemplo, usar POST en lugar de GET).
- `HttpMediaTypeNotSupportedException`: cuando el tipo de contenido de la solicitud no es compatible (por ejemplo, application/xml en lugar de application/json).
- `HttpMessageNotReadableException`: cuando no se puede leer el cuerpo de la solicitud, a menudo debido a un formato JSON inválido.
- `MethodArgumentNotValidException`: cuando la validación de los datos anotados con @Valid falla, a menudo debido a restricciones en los campos.
- `ConstraintViolationException` (de javax.validation): cuando una validación falla en los datos de entrada o en las entidades persistidas, generada principalmente en validaciones personalizadas.
- `EntityNotFoundException`: cuando se busca una entidad que no existe en la base de datos.
- `DataIntegrityViolationException`: cuando se viola una restricción de integridad de datos, como restricciones NOT NULL, UNIQUE, o restricciones de claves foráneas.
- `LazyInitializationException` (de Hibernate): cuando se intenta acceder a una colección lazy-loaded fuera del contexto de una transacción.

Además se podrán manejar todas las excepciones personalizadas

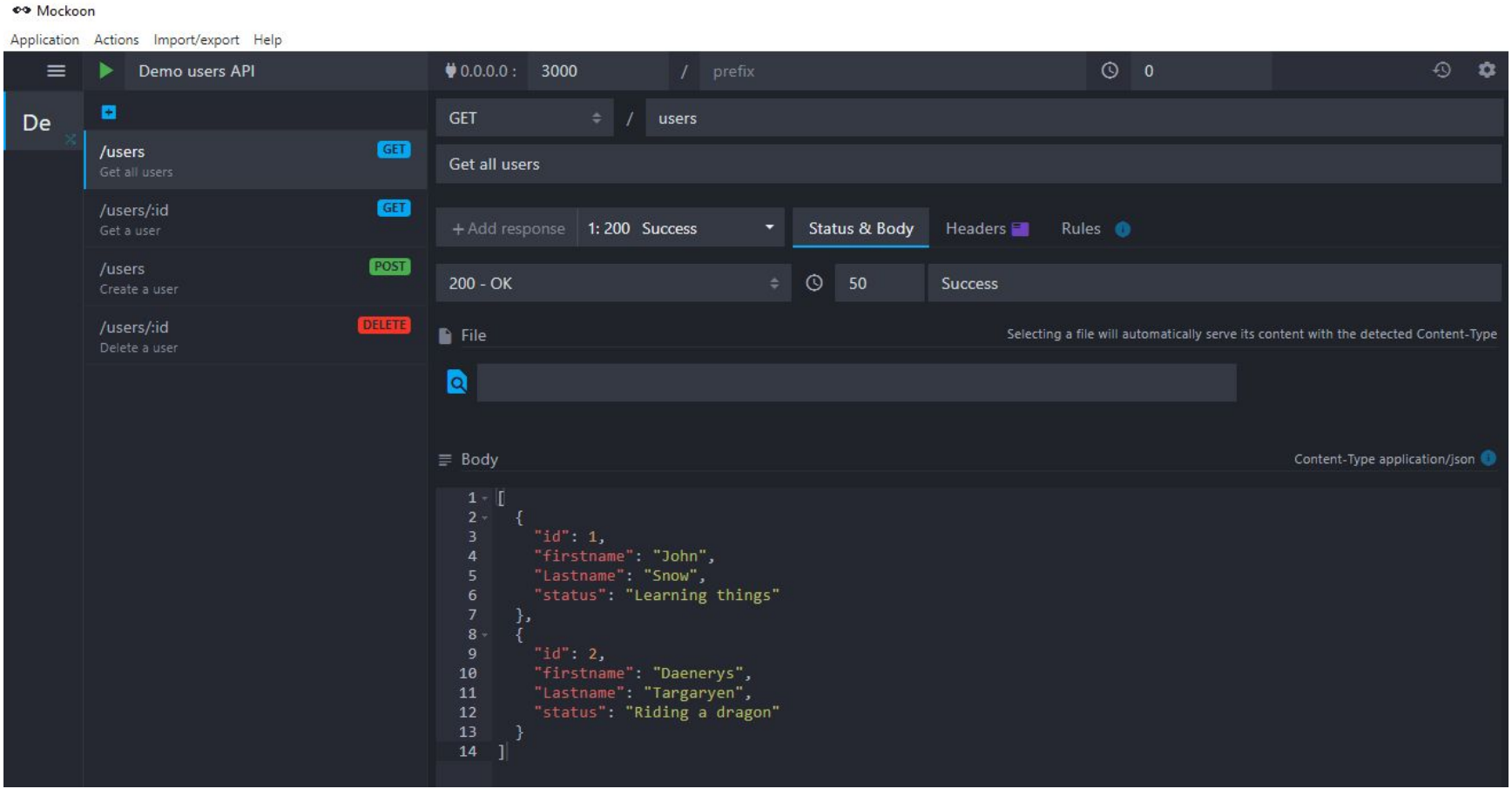
Cliente REST para probar la API: POSTMAN



También como extensión de Chrome o Firefox:



REST Mock server: Mockoon



Sitios para hacer REST Mock APIs