ESCUELA POLITÉCNICA SUPERIOR DE CÓRDOBA
Universidad de Córdoba

UNIVERSIDAD DE CÓRDOBA

Programación web

Prácticas

Semana 9: Diseño e implementación de API (I)

Aurora Ramírez Quesada (aramirez@uco.es)

Departamento de Ciencia de la Computación e Inteligencia Artificial

Universidad de Córdoba

Índice de contenido

Diseño de API REST
Introducción

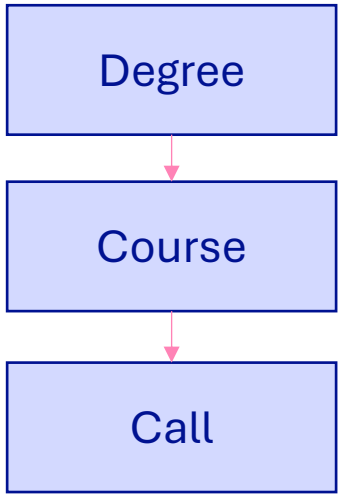
- Arquitectura basada en servicios que utiliza explícitamente los métodos HTTP
- Peticiones y respuestas se transfieren en un formato concreto (JSON, XML, etc.)
- El diseño de una API implica:
  1. Definir las URIs de acceso a los recursos (*endpoints*) como estructura de directorios
  2. Definir controladores que acepten peticiones HTTP (extendemos a PUT, DELETE, etc.)
  3. Proporcionar una implementación coherente con el método HTTP (responsabilidad del programador)
     - Ligeros cambios en la sintaxis de peticiones y respuestas
     - Haciendo uso de patrones de diseño ya utilizados: repositorios y DTOs
  4. Gestionar códigos HTTP para informar del resultado de la operación
     - Es recomendable proporcionar siempre código de estado (OK, *Resource Not Found*, etc.)
     - Cuando sea necesario, enviar la información en el cuerpo de la petición y de la respuesta

Diseño de API REST
Endpoints

- Identificación de los recursos a los que se accederá mediante la API
  - Un recurso es un objeto o colección que puede ser accedido y manipulado a través de la API
  - Se decide qué "exponer" y que no, y con qué métodos soportarlo
  - Para nuestro ejemplo, los recursos son: estudiantes, cursos y profesores
- Podemos especificar el punto de inicio de la API para diferenciarlo de otras direcciones de acceso (como las de MVC)

```
localhost:8080/api/
```

- Notación para acceso a recursos: /api/recurso/{id}
  - Acceso a todos los cursos: */api/courses*
  - Acceso un curso concreto: */api/courses/1*
  - Acceso a recurso anidado: */api/degrees/computer-science/courses/ai/calls/january*



Diseño de API REST
Endpoints

La norma fundamental en el diseño de API REST es **mantener la consistencia**

- Principios de nombrado (buenas prácticas):
  1. Los recursos se corresponden con nombres, no con verbos ni operaciones CRUD: */students* en vez de */getStudents*
  2. Las colecciones se nombran en plural: */students*
  3. Utilizar anidación de recursos para jerarquía de objetos: */students/{id}/address*
  4. Para nombres compuestos, es preferible el guion medio al guion bajo: */store-departments*
  5. No utilizar mayúsculas
  6. No indicar extensiones de formato (si es necesario, utilizar el atributo *content-type*)
  7. Si tenemos varias versiones de la API, diferenciarlas: */api/v1/students*
     - Alternativa como parámetro en la URL: */api/students?version=1*

Fuente: https://restfulapi.net/resource-naming/

Diseño de API REST
Métodos API

- En la práctica 1 solo se soportaban peticiones GET y POST
- Spring establece un tipo de anotación para mapear otros métodos HTTP: PUT, PATCH, DELETE
- El hecho de recibir una petición no implica que se vaya a cumplir su propósito, es el desarrollador quien decide el comportamiento del controlador conforme al propósito esperado
- En **API REST** es incluso más importante respetar la semántica según la anotación

Anotación	Método	Propósito
@GetMapping	HTTP GET	Leer datos de un recurso
@PostMapping	HTTP POST	Crear un recurso
@PutMapping	HTTP PUT	Actualizar (reemplazar) un recurso
@PatchMapping	HTTP PATCH	Actualizar (parte de) un recurso
@DeleteMapping	HTTP DELETE	Borrar recurso
@RequestMapping	Petición general, el tipo de petición se especifica como atributo (“*method*”)	

Diseño de API REST
Códigos de estado

- Las APIs están ideadas para ser consumidas por otros sistemas (clientes), no necesariamente de forma visual (cliente HTML)
- Como parte de la respuesta, se debe informar al cliente del resultado de la operación
- Se recomienda utilizar los códigos de estado de HTTP (RFC 9110):
  - Códigos de estado para casos de éxito: (200 – 299)
  - Códigos de estado para errores de cliente: (400 – 499)
- El significado de un código puede variar según el método HTTP o estar vinculados a uno concreto:
  - Código **200** (*OK*) para GET significa que el recurso se ha podido leer, mientras que con PUT implica que se ha modificado correctamente
  - Código **201** (*Created*) es habitual vincularlo a peticiones POST que terminan satisfactoriamente

https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status
https://httpwg.org/specs/rfc9110.html#overview.of.status.codes

Diseño de API REST
Códigos de estado

- Códigos de estado más comunes

Código	Nombre	Significado
200	*OK*	La petición ha sido exitosa
201	*Created*	El recurso ha sido creado correctamente
204	*No Content*	El recurso ha sido borrado correctamente
400	*Bad Request*	La petición es inválida, por ejemplo, porque faltan datos en el cuerpo de la petición
404	*Not Found*	El recurso buscado no se encuentra
405	*Method Not Allowed*	La petición utiliza un método HTTP no soportado
422	*Unprocessable Entity*	La petición no puede procesarse porque la información enviada no es válida

https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status
https://httpwg.org/specs/rfc9110.html#overview.of.status.codes

API REST en Spring
Introducción

- En Spring, podemos implementar una API REST utilizando los siguientes elementos:
  - **Controladores**: Clase que se encarga de recibir las peticiones asociadas a un recurso. Se indican con la anotación `@RestController`
  - **Variables en la URI**: Con la anotación `@PathVariable` se podrá extraer aquellos identificadores asociados a los recursos que aparecen en la URI
  - **Cuerpo de la petición**: Con la anotación `@RequestBody` podemos acceder al cuerpo de la petición, donde estarán los datos del recurso que se solicita crear o modificar
  - **Respuesta**: La clase `ResponseEntity` permite construir la respuesta JSON a partir de un objeto que se quiera devolver
  - **Códigos de estado**: La clase `HTTPStatus` y la anotación `@ResponseStatus` permiten indicar el código de estado tras procesar la petición

API REST en Spring
Introducción

- Estructura general de los controladores REST para peticiones GET y POST

```
@GetMapping(“/{id}”)
Public ResponseEntity<T> getResourceById(@PathVariable Long id){
        // Procesar petición
        T objectFound = repository.findById(id);
        // Crear respuesta
        ResponseEntity response = new ResponseEntity<>(objectFound, HttpStatus.OK);
        return response;
}
```

GET

```
@PostMapping(consumes=“application/json”)
@ResponseStatus(HttpStatus.CREATED)
Public T postResource(RequestBody objectToCreate){
        // Guardar datos
        T createdObject = repository.save(objectToCreate);
        // Devolver objeto creado
        return createdObject;
}
```

POST

API REST en Spring
Peticiones GET

Por simplicidad, se asume que la búsqueda no va a resultar fallida por problemas de conexión a la base de datos

- **Caso 1:** Acceso a una colección (recurso completo)
  1. La ruta (*path*) de la API se encuentra a nivel de la clase controlador
  2. En este caso, la petición GET no tiene parámetros
  3. Se invoca al repositorio correspondiente para que haga la consulta a base de datos
  4. Se devuelve el objeto iterable junto con el código de estado **OK**

```
@RestController()
@RequestMapping(path="/api/students", produces="application/json")
public class StudentRestController {

    StudentRepository studentRepository;

    @GetMapping
    public ResponseEntity<List<Student>> getAllStudents(){
        List<Student> students = studentRepository.findAllStudents();
        ResponseEntity<List<Student>> response = new ResponseEntity<>(students, HttpStatus.OK);
        return response;
    }
```

API REST en Spring
Peticiones GET

- **Caso 2:** Acceso a una colección parcial (búsqueda parametrizada)
  1. En este caso, la petición GET sí tiene parámetros asociados (*params*)
  2. Los parámetros deben corresponderse con los atributos del método (`@RequestParam`)
  3. Se invoca al repositorio para que haga la consulta a base de datos con los parámetros
  4. Se devuelve el objeto iterable junto con el código de estado **OK**

```
@GetMapping(params="type")
public ResponseEntity<List<Student>> getStudentsByType(@RequestParam String type) {
    List<Student> students = studentRepository.findStudentsByType(StudentType.valueOf(type));
    ResponseEntity<List<Student>> response = new ResponseEntity<>(students, HttpStatus.OK);
    return response;
}
```

Ejemplo de invocación: localhost:8080/api/students?type=FULL_TIME

API REST en Spring
Peticiones GET

- **Caso 3:** Acceso a un recurso concreto (búsqueda por id)

1. El identificador del recurso forma parte de la URI asociada a la petición (`@GetMapping`)
2. El método lo recibe como parámetro (`@PathVariable`)
3. Se invoca al repositorio para que haga la consulta parametrizada a la base de datos
4. La respuesta se crea en función de si el recurso ha sido encontrado (**OK**) o no (**NOT_FOUND**)

```java
@GetMapping("/{id}")
public ResponseEntity<Student> getStudentById(@PathVariable Integer id){
    Student student = studentRepository.findStudentById(id);
    ResponseEntity<Student> response;
    if(student != null){
        response = new ResponseEntity<>(student, HttpStatus.OK);
    }
    else{
        response = new ResponseEntity<>(student, HttpStatus.NOT_FOUND);
    }
    return response;
}
```

Ejemplo de invocación: localhost:8080/api/students/1

API REST en Spring
Peticiones POST

- Petición **POST** (versión 1)
  1. Se indica el formato esperado (JSON)
  2. Se informa del resultado (CREATED)
  3. Spring automatiza el acceso al cuerpo de la petición extrayendo directamente un objeto mapeado (`@RequestBody`)
  4. Si es necesario, el controlador puede hacer cambios en el objeto antes de guardarlo (p. ej., asignarle un ID)
  5. Lo habitual es devolver el objeto que ha sido guardado finalmente (por si ha sido alterado)

```java
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Student postStudent(@RequestBody Student student) {
    boolean resultOk = studentRepository.addStudent(student);
    if(resultOk)
        return student;
    else
        return null;
}
```

Ejemplo de invocación: localhost:8080/api/students

Cuerpo de la petición (*request body*):

```
{
   “id”: “10”,
   “name”: “Gonzalo”,
   “surname”: “Garcia”,
   “birthDate”: “10-10-2002”,
   “type”: “FULL_TIME”
}
```

API REST en Spring
Peticiones POST

- Petición **POST** (versión 2)
  1. El controlador necesita hacer varias comprobaciones antes de guardar la información
  2. El código de respuesta se adapta según la validez de la información enviada:
     - CREATED: El estudiante no existe y tiene la edad mínima
     - UNPROCESSABLE_ENTITY: Datos incorrectos o el estudiante ya existe
     - INTERNAL_SERVER_ERROR: Fallo en la conexión a la base de datos o al ejecutar la consulta

```java
@PostMapping(consumes="application/json")
public ResponseEntity<Student> postStudent(@RequestBody Student student) {
    ResponseEntity<Student> response;
    Integer idStudent = studentRepository.getStudentIdIfExists(student.getName(), student.getSurname());
    if(idStudent ==-1){
        if(student.getBirthDate().getYear() > 2009){
            response = new ResponseEntity<>(student, HttpStatus.UNPROCESSABLE_ENTITY);
        }
        else{
            int nextId = studentRepository.findAllStudents().size() + 1;
            student.setId(nextId);
            boolean resultOk = studentRepository.addStudent(student);
            if(resultOk){
                response = new ResponseEntity<>(student, HttpStatus.CREATED);
            }
            else{
                response = new ResponseEntity<>(student, HttpStatus.INTERNAL_SERVER_ERROR);
            }
        }
    }
    else{
        response = new ResponseEntity<>(student, HttpStatus.UNPROCESSABLE_ENTITY);
    }
    return response;
}
```

API REST en Spring
Código cliente

- La clase `RestTemplate` de Spring permite realizar invocaciones a la API desde Java
  - Métodos para crear peticiones HTTP (GET, POST, etc.) indicando la URL como `String`
  - Permite recuperar la respuesta como objeto `ResponseEntity`
  - Ofrece mecanismos de conversión para manejar objetos Java en lugar de datos en JSON

Método	Propósito
`getForEntity`	Envía una petición GET devolviendo un objeto `ResponseEntity`
`getForObject`	Envía una petición GET transformando la respuesta en un objeto Java
`postForEntity`	Envía una petición POST devolviendo un objeto `ResponseEntity`
`postForObject`	Envía una petición POST transformando la respuesta en un objeto Java
`execute`	Ejecuta una petición HTTP según el método especificado, y devuelve un objeto

API REST en Spring
Código cliente

- Enviar peticiones **GET**
  - Si solo queremos recuperar el objeto: `getForObject("url", JavaClass, uriVariables)`
  - Si interesan detalles de la respuesta: `getForEntity("url", JavaClass, uriVariables)`

```
RestTemplate rest = new RestTemplate();
String baseURL = "http://localhost:8080";

// Request to retrieve all students
ResponseEntity<Student[]> response = rest.getForEntity(baseURL + "/api/students", responseType: Student[].class);
List<Student> listOfStudents = Arrays.asList(response.getBody());
System.out.println(x: "==== REQUEST 1: GET all students ====");
Date date = new Date(response.getHeaders().getDate());
System.out.println("Response date: " + date);
for(Student s: listOfStudents){
    System.out.println(s);
    System.out.println(x: "--------------");
}
```

GET localhost:8080/api/students

```
==== REQUEST 1: GET all students ====
Response date: Wed Nov 05 13:14:54 CET 2025
        ID: 1
        Name: Andrea
        Surname: Aguirre
        Birth date: 2000-01-10
        Type: FULL_TIME
--------------
        ID: 2
        Name: Beatriz
        Surname: Benitez
        Birth date: 2000-02-12
        Type: FULL_TIME
--------------
```

API REST en Spring
Código cliente

- Enviar peticiones **GET**
  - Si solo queremos recuperar el objeto: `getForObject("url", JavaClass, uriVariables)`
  - Si interesan detalles de la respuesta: `getForEntity("url", JavaClass, uriVariables)`

```
// Request to retrive partial-time students
response = rest.getForEntity(baseURL + "/api/students?type=PARTIAL_TIME", responseType: Student[].class);
listOfStudents = Arrays.asList(response.getBody());
System.out.println(x: "==== REQUEST 2: GET partial-time students ====");
date = new Date(response.getHeaders().getDate());
System.out.println("Response date: " + date);
for(Student s: listOfStudents){
    System.out.println(s);
    System.out.println(x: "--------------");
}
```

GET localhost:8080/api/students?type=PARTIAL_TIME


```
==== REQUEST 2: GET partial-time students ====
Response date: Wed Nov 05 13:14:54 CET 2025
        ID: 3
        Name: Carlos
        Surname: Castro
        Birth date: 2000-03-03
        Type: PARTIAL_TIME
--------------
        ID: 7
        Name: Laura
        Surname: Lopez
        Birth date: 2003-06-18
        Type: PARTIAL_TIME
--------------
```

API REST en Spring
Código cliente

- Enviar peticiones **GET**
  - Si solo queremos recuperar el objeto: `getForObject("url", JavaClass, uriVariables)`
  - Si interesan detalles de la respuesta: `getForEntity("url", JavaClass, uriVariables)`

```
// Request to retrive one student
Student student = rest.getForObject(baseURL + "/api/students/{id}", responseType: Student.class,...uriVariables: 1);
System.out.println(x: "==== REQUEST 3: GET student with id ====");
System.out.println(student.toString());
```

GET localhost:8080/api/students/1


```
==== REQUEST 3: GET student with id ====
        ID: 1
        Name: Andrea
        Surname: Aguirre
        Birth date: 2000-01-10
        Type: FULL_TIME
```

API REST en Spring
Código cliente

- Enviar peticiones **POST**
  - Si solo queremos recuperar el objeto: `postForObject("url", JavaClass, uriVariables)`
  - Si interesan detalles de la respuesta: `postForEntity("url", JavaClass, uriVariables)`

POST localhost:8080/api/students

```
RestTemplate rest = new RestTemplate();
String baseURL = "http://localhost:8080";

// POST a new student (valid)
LocalDate birthDate = LocalDate.of(year: 2002, month: 10, dayOfMonth: 10);
Student newStudent = new Student(-1, name: "Gonzalo", surname: "Garcia", birthDate, StudentType.FULL_TIME);
ResponseEntity<Student> response;

try{
    response = rest.postForEntity(baseURL + "/api/students", newStudent, responseType: Student.class);
    System.out.println(x: "==== REQUEST 4: POST student (valid) ====");
    System.out.println("Status code: " + response.getStatusCode());
    System.err.println("Response body:\n" + response.getBody());
}catch(HttpClientErrorException exception){
    System.out.println(exception);
}
```

```
==== REQUEST 4: POST student (valid) ====
Status code: 201 CREATED
Response body:
        ID: 8
        Name: Gonzalo
        Surname: Garcia
        Birth date: 2002-10-10
        Type: FULL_TIME
```

API REST en Spring
Código cliente

- Enviar peticiones **POST**
  - Si solo queremos recuperar el objeto: `postForObject("url", JavaClass, uriVariables)`
  - Si interesan detalles de la respuesta: `postForEntity("url", JavaClass, uriVariables)`

POST localhost:8080/api/students

```
// POST a student (invalid)
birthDate = LocalDate.of(year: 2012, month: 10, dayOfMonth: 10);
newStudent.setName(name: "Mathew");
newStudent.setSurname(surname: "Murphy");
newStudent.setType(StudentType.ERASMUS);
newStudent.setBirthDate(birthDate);

System.out.println(x: "==== REQUEST 5: POST student (invalid) ====");
try{
    response = rest.postForEntity(baseURL + "/api/students", newStudent, responseType: Student.class);
}catch(HttpClientErrorException exception){
    System.out.println(exception);
}
```

```
==== REQUEST 5: POST student (invalid) ====
org.springframework.web.client.HttpClientErrorException$UnprocessableEntity: 422  on POST request for "http://localhost:8080/api/students":
"{"id":-1,"name":"Mathew","surname":"Murphy","birthDate":"2012-10-10","type":"ERASMUS"}"
```

Objetivos de la semana

1. Replica el ejemplo explicado en clase, ejecutándolo sobre tu base de datos
   - Versión completa disponible en el repositorio Github: https://github.com/aramirez-uco/pw-examples
2. Comienza a diseñar los *endpoints* de tu API según el enunciado de la práctica 2
   - Aborda solo las funciones de lectura (petición GET) y creación (petición POST)
3. Comienza a implementar la API
   - Define un controlador por recurso, con las anotaciones apropiadas para los métodos GET y POST
   - Devuelve códigos HTTP apropiados según el resultado de la operación
   - Implementa código cliente para probar la API
4. Consulta la bibliografía recomendada:
   - Secciones 7.1 y 7.3 del capítulo 7: "*Creating REST services*" del libro "*Spring in Action*" (6th ed.)
   - Tutorial Spring REST (avanzado): https://spring.io/guides/tutorials/rest