Practica 3:

**HTTP**

● ¿Qué es y cómo funciona el protocolo HTTP?

<https://www.w3.org/Protocols/HTTP/1.1/draft-ietf-http-v11-spec-01>

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990.

**Overall Operation**

The HTTP protocol is based on a request/response paradigm. A client establishes a connection with a server and sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible body content.

Most HTTP communication is initiated by a user agent and consists of a request to be applied to a resource on some origin server. In the simplest case, this may be accomplished via a single connection (v) between the user agent (UA) and the origin server (O).

request chain ------------------------>

UA -------------------v------------------- O

<----------------------- response chain

A more complicated situation occurs when one or more intermediaries are present in the request/response chain. There are three common forms of intermediary: proxy, gateway, and tunnel

request chain -------------------------------------->

UA -----v----- A -----v----- B -----v----- C -----v----- O

<------------------------------------- response chain

The figure above shows three intermediaries (A, B, and C) between the user agent and origin server. A request or response message that travels the whole chain must pass through four separate connections.

On the Internet, HTTP communication generally takes place over TCP/IP connections. The default port is TCP 80 [19], but other ports can be used. This does not preclude HTTP from being implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport; any protocol that provides such guarantees can be used, and the mapping of the HTTP/1.1

● Mencionar los métodos o verbos de HTTP: GET, POST, DELETE, PUT, OPTIONS

<https://www.w3.org/Protocols/HTTP/1.1/draft-ietf-http-v11-spec-01#Methods>

The **GET** method means retrieve whatever information (in the form of an entity) is identified by the Request-URI.

The **HEAD** method is identical to GET except that the server MUST NOT return a message-body in the response. The metainformation contained in the HTTP headers in response to a HEAD request SHOULD be identical to the information sent in response to a GET request.

The **PUT** method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server.

The **DELETE** method requests that the origin server delete the resource identified by the Request-URI.

The **OPTIONS** method represents a request for information about the communication options available on the request/response chain identified by the Request-URI.OPTIONS shows users which HTTP methods are available for a specific URL

PATCH partially modifies a web resource. PATCH is no longer available on the w3.org site.

● ¿Los request de tipo POST tiene un body? ¿Y los de tipo GET?

POST – Yes

GET - No

● Códigos de estado de response HTTP: Describa los código: 200, 201, 400, 401, 403 y 500

<https://www.w3.org/Protocols/HTTP/1.1/draft-ietf-http-v11-spec-01#Status-Codes>

### Successful 2xx

200 OK - The request has succeeded.

201 - Created The request has been fulfilled and resulted in a new resource being created.

### Client Error 4xx

400 Bad Request - The request could not be understood by the server due to malformed syntax

404 Not Found - The server has not found anything matching the Request-URI.

403 Forbidden - The server understood the request, but is refusing to fulfill it. Authorization will not help and the request should not be repeated.

### Server Error 5xx

500 Internal Server Error - The server encountered an unexpected condition which prevented it from fulfilling the request.

**SPRING.IO**

Spring Web MVC:

○ Describa el propósito de las anotaciones @Controller y @ResquestMapping

<https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html>

The @Controller annotation indicates that a particular class serves the role of a controller. Spring does not require you to extend any controller base class or reference the Servlet API. However, you can still reference Servlet-specific features if you need to.

The @Controller annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotations.

You use the @RequestMapping annotation to map URLs such as /appointments onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.

The following example from the Petcare sample shows a controller in a Spring MVC application that uses this annotation:

@Controller

@RequestMapping("/appointments")

public class AppointmentsController {

private final AppointmentBook appointmentBook;

@Autowired

public AppointmentsController(AppointmentBook appointmentBook) {

this.appointmentBook = appointmentBook;

}

@RequestMapping(method = RequestMethod.GET)

public Map<String, Appointment> get() {

return appointmentBook.getAppointmentsForToday();

}

@RequestMapping(value="/{day}", method = RequestMethod.GET)

public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {

return appointmentBook.getAppointmentsForDay(day);

}

@RequestMapping(value="/new", method = RequestMethod.GET)

public AppointmentForm getNewForm() {

return new AppointmentForm();

}

@RequestMapping(method = RequestMethod.POST)

public String add(@Valid AppointmentForm appointment, BindingResult result) {

if (result.hasErrors()) {

return "appointments/new";

}

appointmentBook.addAppointment(appointment);

return "redirect:/appointments";

}

}

The first usage is on the type (class) level, which indicates that all handling methods on this controller are relative to the /appointments path.

A @RequestMapping on the class level is not required.

@Controller

public class ClinicController {

private final Clinic clinic;

@Autowired

public ClinicController(Clinic clinic) {

this.clinic = clinic;

}

@RequestMapping("/")

public void welcomeHandler() {

}

@RequestMapping("/vets")

public ModelMap vetsHandler() {

return new ModelMap(this.clinic.getVets());

}

○ Menciones que otras anotaciones complementan a ResquestMapping para machear los distintos métodos de HTTP.

○ ¿Qué permiten las anotaciones @RequestParam, @ResquestHeader, @PathVariable, @RequestBody.?

<https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html>

**URI Template Patterns**

URI templates can be used for convenient access to selected parts of a URL in a @RequestMapping method. A URI Template is a URI-like string, containing one or more variable names. A template variable is enclosed with curly braces.

In Spring MVC you can use the @PathVariable annotation on a method argument to bind it to the value of a URI template variable:

@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)

public String findOwner(@PathVariable String ownerId, Model model) {

Owner owner = ownerService.findOwner(ownerId);

model.addAttribute("owner", owner);

return "displayOwner";

}

A method can have any number of @PathVariable annotations:

@RequestMapping(value="/owners/{ownerId}/pets/{petId}", method=RequestMethod.GET)

public String findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {

Owner owner = ownerService.findOwner(ownerId);

Pet pet = owner.getPet(petId);

model.addAttribute("pet", pet);

return "displayPet";

}

**Binding request parameters to method parameters with @RequestParam**

Use the @RequestParam annotation to bind request parameters to a method parameter in your controller.

'value' element of @RequestParam is used to specify URL query param name. Following handler method will be mapped with the request /employees?dept=IT :

Controller

@RequestMapping("employees")

public class EmployeeController {

@RequestMapping

public String handleEmployeeRequestByDept (@RequestParam("dept") String deptName,

Model map) {

map.addAttribute("msg", "employees request by dept: " + deptName);

return "my-page";

}

}

**Using multiple @RequestParam annotations**

A method can have any number of @RequestParam annotations. Following will be mapped with /employees?dept=IT&state=NC :

@Controller

@RequestMapping("employees")

public class EmployeeController {

@RequestMapping

public String handleEmployeeRequestByDept (@RequestParam("dept") String deptName,

@RequestParam("state") String stateCode,

Model map) {

map.addAttribute("msg", "employees request by dept and state code : "+

deptName+", "+stateCode);

return "my-page";

}

}

**Mapping the request body with the @RequestBody annotation**

The @RequestBody method parameter annotation indicates that a method parameter should be bound to the value of the HTTP request body. Spring will bind the incoming HTTP request body(for the URL mentioned in @RequestMapping for that method) to that parameter. Spring will [behind the scenes] use HTTP Message converters to convert the HTTP request body into domain object

@RequestMapping(value="/user/create", method=RequestMethod.POST)

public ResponseEntity<Void> createUser(@RequestBody User user, UriComponentsBuilder ucBuilder){

System.out.println("Creating User "+user.getName());

//….

}

**Mapping request header attributes with the @RequestHeader annotation**

The @RequestHeader annotation allows a method parameter to be bound to a request header.

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RequestMapping;

//..

@Controller

public class HelloController {

@RequestMapping(value = "/hello.htm")

public String hello(@RequestHeader(value="User-Agent") String userAgent)

//..

}

}

When spring maps the request, it checks http header with name “User-Agent” and bind its value to String userAgent. If the header value that you specified does not exists in request, Spring will initialize the parameter with null value.

○ ¿Cuál es el propósito de la anotación @Service?

Service Components are the class file which contains @Service annotation. These class files are used to write business logic in a different layer.

The logic for creating a service component class file is shown here −

public interface ProductService {

public abstract void createProduct(Product product);

public abstract void updateProduct(String id, Product product);

public abstract void deleteProduct(String id);

public abstract Collection<Product> getProducts();

}

@Service

public class ProductServiceImpl implements ProductService {

private static Map<String, Product> productRepo = new HashMap<>();

static {

Product honey = new Product();

honey.setId("1");

honey.setName("Honey");

productRepo.put(honey.getId(), honey);

Product almond = new Product();

almond.setId("2");

almond.setName("Almond");

productRepo.put(almond.getId(), almond);

}

@Override

public void createProduct(Product product) {

productRepo.put(product.getId(), product);

}

@Override

public void updateProduct(String id, Product product) {

productRepo.remove(id);

product.setId(id);

productRepo.put(id, product);

}

@Override

public void deleteProduct(String id) {

productRepo.remove(id);

}

@Override

public Collection<Product> getProducts() {

return productRepo.values();

}

}

@Controller

public class ProductServiceController {

@Autowired

ProductService productService;

@RequestMapping(value = "/products")

public ResponseEntity<Object> getProduct() {

return new ResponseEntity<>(productService.getProducts(), HttpStatus.OK);

}

@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)

public ResponseEntity<Object>

updateProduct(@PathVariable("id") String id, @RequestBody Product product) {

productService.updateProduct(id, product);

return new ResponseEntity<>("Product is updated successsfully", HttpStatus.OK);

}

@RequestMapping(value = "/products/{id}", method = RequestMethod.DELETE)

public ResponseEntity<Object> delete(@PathVariable("id") String id) {

productService.deleteProduct(id);

return new ResponseEntity<>("Product is deleted successsfully", HttpStatus.OK);

}

@RequestMapping(value = "/products", method = RequestMethod.POST)

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

productService.createProduct(product);

return new ResponseEntity<>("Product is created successfully", HttpStatus.CREATED);

}

}

○ Para que se usa la anotación @Autowired

@Autowired is used to Dependency Injection

● Spring-boot: Explique brevemente el propósito de este projecto de Spring

**Ejercicio práctico**

1. Instale la Spring Tool Suite 4 (<https://spring.io/tools>)
2. Seleccione el workspace en la carpeta de versionado de GIT (previamente debio crear un repositorio en GITHub y vincularlo con el repositorio local)
3. Luego, cree un proyecto con la opción File, New…
4. Seleccione la opción Spring Starter Project
5. Seleccione la dependencias del Proyecto Spring Boot DevTools, Spring Web, Spring Data JPA(no si no tiene instalado el motor de BD) y el driver JDBC para su base de datos(no si no tiene instalado el motor de BD).