4. Acceso a servicios desde Java

Parte II. Comandos, autenticación y carga de archivos

Programación de Servicios y Procesos

Nacho Iborra
Álvaro Pérez
Javier Carrasco

Tabla de contenido

1. GET, POST, PUT y DELETE

Las 4 operaciones comunes en los servicios web REST son GET, POST, PUT y DELETE, como se vio en la unidad anterior. Si bien puedes hacer todo solo con GET y POST y llamar a diferentes servicios para diferentes acciones (añadir, actualizar o eliminar un elemento, por ejemplo), es recomendable, y más limpio, acceder al mismo servicio web (o recurso) y especificar la operación que se desea hacer.

Como se dijo antes, los métodos más utilizados son GET (datos en la URL y de longitud limitada) y POST (datos enviados en el cuerpo de la solicitud, pueden ser mucho más grandes), pero eso depende de cómo estén configurados los servicios web. A continuación, se verá cómo interactuar con los servicios web que admiten estas operaciones en Java:

1.1. Introducción a la aplicación de ejemplo: *"Product Manager"*

Para ayudar con la explicación, se utilizará una aplicación llamada *"Product Manager"* que simula el catálogo de una tienda de informática, de manera que los productos se almacenan en un servidor con una base de datos *MongoDB* y las operaciones con esos productos se realizarán a través de servicios web en *Express.js*. En la base de datos habrá una colección de productos y una colección de categorías (cada producto pertenece a una categoría determinada):

También puedes verificar la estructura de ambas colecciones (*categories* y *products*) verificando los archivos fuente del modelo en la sub-carpeta *"models"* del proyecto.

Deberías poder descargar todo (proyecto de aplicación JavaFX y proyecto de Node para el servidor, incluido un generador de base de datos) desde el aula virtual para probar esta aplicación. Puedes volver a generar la base de datos cuando lo desees con el generador de base de datos llamado *"db_generator.js"* dentro del proyecto Node.

```
node db_generator.js
```

1.2. GET

Las operaciones **GET** están destinadas a recuperar objetos existentes de la base de datos (SELECT). En este caso, existen 2 servicios web:

- Un servicio que utiliza el recurso *"/category"* que obtiene todas las categorías de la base de datos. Así es como se ve en Node con *Express*:

```
app.get('/category', (req, res) => {
    Category.find().then(result => {
        res.send(result);
    }).catch(error => {
```

```
        res.send([]);
    });
});
```

- Un servicio que utiliza el recurso *"/product/:category"* que recibe un ID de categoría y devuelve todos los productos que pertenecen a esa categoría.

```
app.get('/product/:idCat', (req, res) => {
    Product.find({category: req.params.idCat}).then(result => {
        res.send(result);
    }).catch (error => {
        res.send([]);
    });
});
```

Cuando se recibe una lista de objetos de Java, se necesita usar dos clases especiales llamadas ***com.google.gson.reflect.TypeToken*** y ***java.lang.reflect.Type*** si se pretende usar la biblioteca GSON. Este es el servicio JavaFX que obtendrá la lista de categorías:

```
public class GetCategories extends Service<List<Category>> {
    @Override
    protected Task<List<Category>> createTask() {
        return new Task<List<Category>>() {
            @Override
            protected List<Category> call() throws Exception {
                String json = ServiceUtils.getResponse(
                        "http://localhost:8080/category", null, "GET");
                Gson gson = new Gson();
                Type type = new TypeToken<List<Category>>() {
                }.getType();
                List<Category> cats = gson.fromJson(json, type);
                return cats;
            }
        };
    }
}
```

Y este es el servicio que obtendrá los productos de una categoría:

```
public class GetProducts extends Service<List<Product>> {
    String catId;

    public GetProducts(String catId) {
        this.catId = catId;
    }

    @Override
    protected Task<List<Product>> createTask() {
        return new Task<List<Product>>() {
            @Override
            protected List<Product> call() throws Exception {
                String json = ServiceUtils.getResponse(
                        "http://localhost:8080/product/" + catId,
                        null, "GET");

                Gson gson = new Gson();
                Type type = new TypeToken<List<Product>>() {
                }.getType();
                List<Product> prods = gson.fromJson(json, type);
                return prods;
            }
```

```
        };
    }
}
```

Para finalizar este ejemplo de GET, se verá cómo solicitar productos cuando se selecciona una nueva categoría en la aplicación:

```
private void selectNewCategory(Category category, String selectAfter) {
    getProds = new GetProducts(category.getId());
    getProds.start();
    getProds.setOnSucceeded(e -> {
        currentProds = FXCollections.observableArrayList(getProds.getValue());
        productsTable.setItems(currentProds);
        Optional<Product> selProd = currentProds.stream()
                .filter(p -> p.getId().equals(selectAfter)).findFirst();
        if(selProd.isPresent()) {
            productsTable.getSelectionModel().select(selProd.get());
            productsTable.scrollTo(selProd.get());
        }
    });
}
```

Ejercicio 4

Crea un proyecto llamado ***GetCompanies*** en JavaFX. Utilizará estos dos servicios web utilizando GET:

- ***http://<server_address>/company*** → Devolverá todas las empresas con este formato JSON:

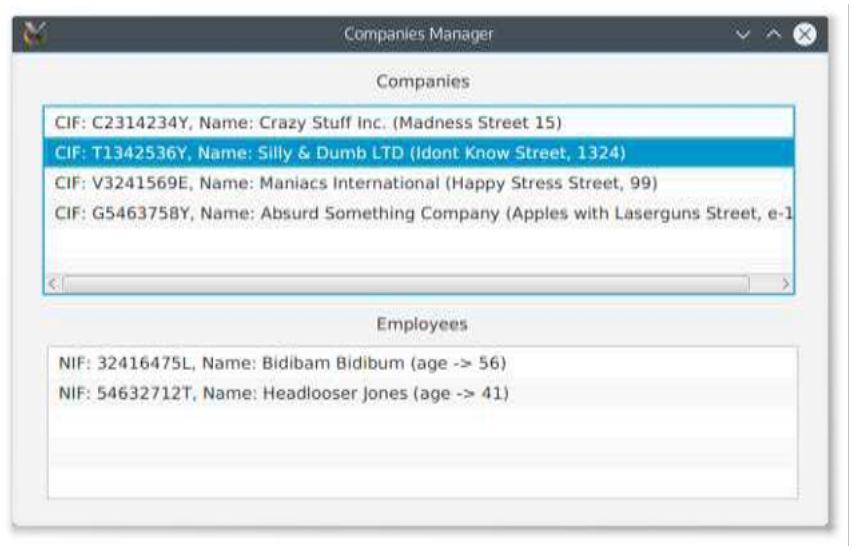
```
[
    {
        "_id": "1asdasrqwwr535q35a",
        "cif": "C2314234Y",
        "name": "Crazy Stuff Inc.",
        "address": "Madness Street 15"
    },
     {
        "_id": "2425ehpasuhrpasueg",
        "cif": "T1342536Y",
        "name": "Silly & Dumb LTD",
        "address": "Idont Know Street, 1324"
     },
    ...
]
```

- ***http://<server_address>/company/{id}*** → Devolverá la información de una empresa en este formato (si hay información que no necesita, como *"idCompany"*, simplemente ignórala y no la incluyas en la clase de Java):

```
{
   "ok": true,
   "error": "",
   "company": {
       "_id": "348w9ueasd90ays8s",
       "cif": "V3241569E",
       "name": "Maniacs International",
       "address": "Happy Stress Street, 99",
       "employees": [
```

```
            {
                "_id": "5sdaoishdaps8ys",
                "nif": "46374869U",
                "name": "Cocaine Rupert",
                "age": "37",
                "company": "348w9ueasd90ays8s"
            },
            {
                "_id": "sdasd6asdas8y8fays",
                "nif": "12425364K",
                "name": "Happysad Windows",
                "age": "47",
                "company": "348w9ueasd90ays8s"
            }
        ]
    }
}
```

Primero, se cargarán todas las empresas en la lista superior del servicio web apropiado. Luego, cuando un usuario seleccione una empresa, recuperará su información y empleados del otro servicio web, mostrándolos en la lista inferior. Esta aplicación se verá más o menos así:



Se te proporcionará un servidor Node con los servicios ya implementados. También incluye un archivo llamado *db_generator.js* que puedes (debes) ejecutar para crear y llenar la base de datos (ejecútalo con *node db_generator.js*). Luego, inicia el servidor (archivo *app.js*) y comienza a crear tu cliente JavaFX.

1.3. POST

Mediante POST, los datos se envían en el cuerpo de la solicitud. Esto se hace utilizando el flujo de salida de la conexión para enviar esos datos. El formato puede ser una URL (*key=value&key2=value2*) o como en este caso, JSON (en formato *String*).

En el ejemplo *product manager*, se ha definido un servicio web que obtiene los datos sobre un producto, procesa la información e inserta este producto en la base de datos. Devuelve (imprime) la identificación del producto recién creado (o una cadena vacía si hubo un error).

Este es el servicio en Java que llamará a este servicio web y devolverá su respuesta:

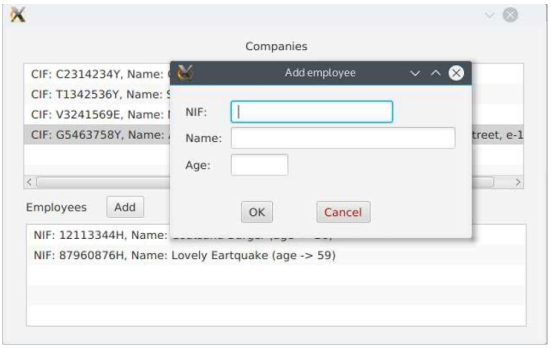
```
public class AddProduct extends Service<String> {
    Product prod;

    public AddProduct(Product prod) {
        this.prod = prod;
    }

    @Override
    protected Task<String> createTask() {
        return new Task<String>() {
            @Override
            protected String call() throws Exception {
                Gson gson = new Gson();
                String resp = ServiceUtils.getResponse(
                        "http://localhost:8080/product",
                        gson.toJson(prod), "POST");
                return resp;
            }
        };
    }
}
```

Finalmente, este será el código en el controlador de la vista para iniciar el servicio y procesar su resultado:

```
addProd = new AddProduct(newProd);
addProd.start();

addProd.setOnSucceeded(e -> {
    String id = addProd.getValue();

    if(!id.equals("")) { // Success
        selectNewCategory(categories.stream()
                .filter(c -> c.getId().equals(newProd.getIdCategory()))
                .findFirst().get(), id);
        showInfoMsg("New product added successfully.", false);
    } else {
        showInfoMsg("Error adding the product", true);
    }
});
```

Ejercicio 5

Actualiza el proyecto ***GetCompanies*** del ejercicio anterior y añade un botón *"Add"* para agregar empleados. Este botón (solo activo cuando se selecciona una empresa) abrirá una nueva ventana que contendrá un formulario para añadir un nuevo empleado.



Para crear una nueva ventana usando otro FXML (o construyendo la escena por código), puedes hacerlo así:

```
Stage stage = new Stage();
stage.initModality(Modality.APPLICATION_MODAL);

Parent root = FXMLLoader.load(getClass().getResource("AddEmployee.fxml"));
Scene scene = new Scene(root);

stage.setTitle("Add employee");
stage.setScene(scene);
stage.show();
stage.setOnHidden((e) -> {
    // Update the employees list somehow…
});
```

Para cancelar y cerrar la ventana abierta (al menos usando FXML):

```
Stage window = (Stage) ((Node) event.getSource()).getScene().getWindow();
window.close();
```

Este formulario deberá ser enviado a este servicio web por POST:

```
http://<server_address>/employee/{idCompany}
```

Y deberá enviar esta información (puede haber más campos y se ignorarán):

```
{
    "age": 32,
    "nif": "12324354T",
    "name": "Delete Meplease"
}
```

Este servicio web devolverá una bandera booleana que indicará si todo salió bien o no, y la identificación del nuevo empleado si todo está correcto, o un mensaje de error si algo salió mal:

```
{
    "ok": true,
    "id": "1sdasp8yawasa8s2"
}

{
    "ok": false,
    "error": "Error adding the employee"
}
```

1.4. PUT

Este método HTTP es una mezcla entre GET y POST, y generalmente es equivalente a la instrucción UPDATE en SQL. La información para obtener el objeto (*id*, por ejemplo) que se modificará en la base de datos se envía en la URL (como GET), y los datos para modificar de ese objeto se envían en el cuerpo (como POST).

En el ejemplo *product manager*, hay un servicio web que procesará la información y actualizará un producto existente. Simplemente devuelve *true* o *false* indicando si la operación fue correcta o no. Este es el servicio JavaFX creado para conectarse a este servicio web:

```
public class UpdateProduct extends Service<Boolean> {
    Product prod;

    public UpdateProduct(Product prod) {
        this.prod = prod;
    }

    @Override
    protected Task<Boolean> createTask() {
        return new Task<Boolean>() {

            @Override
            protected Boolean call() throws Exception {
                Gson gson = new Gson();
                String resp = ServiceUtils.getResponse(
                        "http://localhost:8080/product/" + prod.getId(),
                        gson.toJson(prod), "PUT");
                return Boolean.parseBoolean(resp);
            }
        };
    }
}
```

Y finalmente, así es como se creará e iniciará el servicio desde el controlador:

```
updateProd = new UpdateProduct(newProd);
updateProd.start();

updateProd.setOnSucceeded(e -> {
    if(updateProd.getValue()) { // Success
        selectNewCategory(categories.stream()
                .filter(c -> c.getId().equals(newProd.getIdCategory()))
                .findFirst().get(), newProd.getId());
        showInfoMsg("Product updated successfully.", false);
    } else {
        showInfoMsg("Error updating the product", true);
    }
});
```

1.5. DELETE

La operación DELETE funciona como GET (variables en URL), pero en lugar de devolver objetos que cumplan algunas condiciones, elimina los objetos especificados de la base de datos. Se dispone de un servicio web listo en el ejemplo *product manager*, y este es el servicio JavaFX que se conectará con este servicio web:

```
public class DeleteProduct extends Service<Boolean> {
    String idProd;

    public DeleteProduct(String idProd) {
        this.idProd = idProd;
    }

    @Override
    protected Task<Boolean> createTask() {
        return new Task<Boolean>() {
            @Override
            protected Boolean call() throws Exception {
                Gson gson = new Gson();
                String resp = ServiceUtils.getResponse(
                        "http://localhost:8080/product/" + idProd,
                        null, "DELETE");
                return Boolean.parseBoolean(resp);
            }
        };
    }
}
```

Y finalmente, cómo se creará e iniciará el servicio desde el controlador:

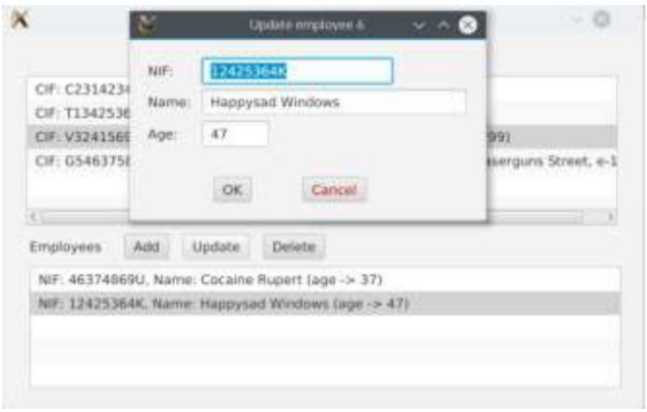
```
deleteProd = new DeleteProduct(selectedProd.getId());
deleteProd.start();

deleteProd.setOnSucceeded(e -> {
    if(deleteProd.getValue()) { // Success
        currentProds.remove(selectedProd);
        showInfoMsg("Product " + selectedProd.getReference() + " deleted.", false);
    } else {
        showInfoMsg("Error removing the product", true);
    }
});
```

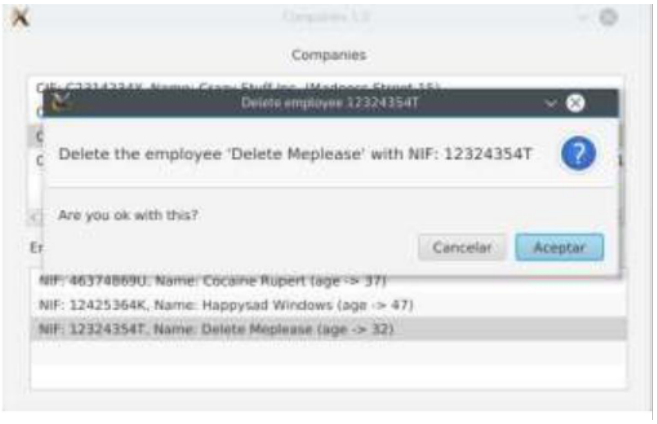
Ejercicio 6

Actualiza el proyecto ***GetCompanies*** de ejercicios anteriores. Deberás añadir dos botones más:

- ***Update:*** Solo activo cuando se selecciona un empleado. Se abrirá una ventana con un formulario (similar a *Add*) para editar la información de un empleado. Se conectará al servicio web y enviará información mediante el método PUT:
  - **http://<server_address>/employee/{id}**
  - La información enviada en JSON tendrá el mismo formato que al añadir un empleado (ver ejercicio 5), y la respuesta será un objeto JSON con dos campos → *"ok"* (booleano) y *"error"* (*String*).

- ***Delete:*** Abrirá un cuadro de diálogo para preguntar al usuario si desea eliminar al empleado seleccionado (ver el ejemplo de *ProductManager*). Si se da una respuesta positiva, llamará a este servicio web (usando el método DELETE):
  - **http://<server_address>/employee/{id}**
  - La respuesta será un objeto JSON con el mismo formato que antes (actualización).





2. Autenticación y carga de ficheros

2.1. Gestionar autenticación mediante *token*

Has aprendido sobre la autenticación de *tokens* en la unidad anterior, por lo que este es un término con el que debes estar familiarizado. La forma estándar de enviar un *token* es en un encabezado llamado *"Authorization"* (aunque podría ser diferente) con el prefijo *"Bearer"* antes del *token* codificado (aunque depende de ti definir este prefijo, siempre que estés encargado de implementar ambos lados, cliente y servidor). Así es como se añadiría este encabezado a una solicitud a través de un objeto *HttpURLConnection* (*conn*):

```
if(token != null) {
    conn.setRequestProperty("Authorization", "Bearer " + token);
}
```

Ya tiene este código añadido a tu clase *ServiceUtils*, junto con un par de métodos útiles: *setToken()* y *removeToken()*, que permiten configurar o eliminar el *token* de las solicitudes realizadas por esta clase.

2.2. Envío de archivos

También has oído hablar de enviar imágenes codificadas en formato ***Base64*** desde el cliente al servidor en la unidad anterior, y cómo procesar esta imagen en el servidor. Para obtener una imagen y convertir sus bytes en formato *Base64* en el lado del cliente, es posible que debas hacer algo como esto:

1. Crear una clase para almacenar la información de la imagen: al menos, el nombre del archivo de imagen y los datos codificados:

```
class MyImage {
    String name;
    String data;

    public MyImage(Path file) {
        name = file.getFileName().toString();
        byte[] bytes;
        data = "";

        try {
            bytes = Files.readAllBytes(file);
            data = Base64.getEncoder().encodeToString(bytes);
        } catch (IOException ex) {
            System.err.println("Error getting bytes from " + file.toString());
        }
    }
}
```

Como puedes ver, se utiliza la clase ***Base64*** de la API de Java (*package java.util*) para codificar los bytes de la imagen.

2. Utiliza esta clase para codificar el archivo de imagen deseado y enviarlo al servicio correspondiente. En este ejemplo, se codifica una imagen llamada *"image.jpg"* de la carpeta actual y se envía a un servicio llamado */uploadImg*.

```
MyImage img = new MyImage(Paths.get("image.jpg"));
Gson gson = new Gson();
String json = gson.toJson(img, MyImage.class);
String resp = ServiceUtils.getResponse("http://localhost:8080/uploadImg", json, "POST");
```

Ejercicio 7

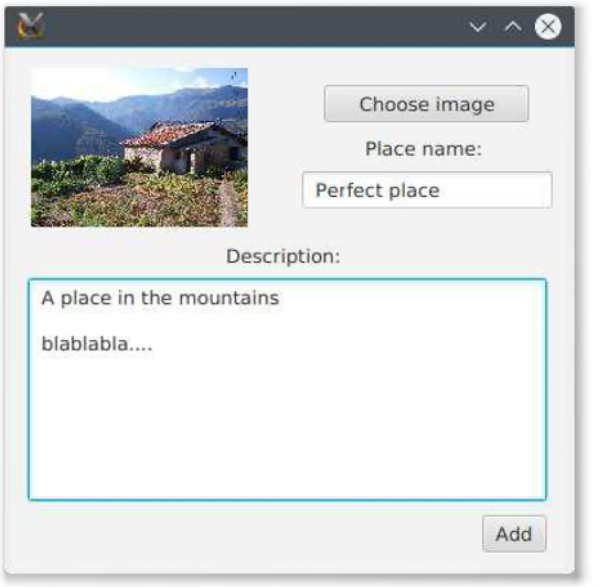
Crea un proyecto JavaFX llamado ***PhotoUploader*** que cargará una foto con un título y una descripción a un servicio web usando POST:

http://<server_address>/photo

El objeto de respuesta JSON contendrá:

"ok" → booleano y *"error"* → *String*.

Este será el formato JSON que tendrás que enviar:




```
{
    "name": "IMG_20130831_174624.jpg",
    "title": "Lovely place",
    "desc": "place description",
    "data": "base 64 encoded data…."
}
```

Accede a esta dirección en tu navegador para verificar todas las imágenes cargadas y eliminarlas: ***http://<server_address>/photoplaces/***