

## *Solutions proposées TP*

### *BDA : TP3*

*Enseigné par :*

**Samir YUCEF**

*Réalisé par l'étudiant :*

- Lucas ZHENG

[lucas.zheng@edu.univ-paris13.fr](mailto:lucas.zheng@edu.univ-paris13.fr)

# Table des questions

Configuration.....	2
Hello World.....	4
Spring Web.....	5
H2 Database et JPA.....	13

# Configuration

Entrez dans le site <https://start.spring.io/>

Ajouter les 3 dépendances avec le bouton "ADD DEPENDENCIES" :

- Spring Web
- Spring Data JPA
- H2 Database

The screenshot shows the Spring Initializr web application interface. The page is dark-themed and contains several sections for configuring a new Spring project. On the left, there are tabs for 'Project' and 'Language'. Under 'Project', 'Maven' is selected. Under 'Language', 'Java' is selected. Below these, 'Spring Boot' versions are listed, with '3.4.4' selected. The 'Project Metadata' section includes fields for 'Group' (com.example), 'Artifact' (BDA\_TP3), 'Name' (BDA\_TP3), 'Description' (Demo project for Spring Boot), and 'Package name' (com.example.BDA\_TP3). There are also options for 'Packaging' (Jar selected) and 'Java' version (21 selected). On the right, the 'Dependencies' section lists 'Spring Data JPA' (SQL), 'Spring Web' (WEB), and 'H2 Database' (SQL), each with a description. At the bottom, there are buttons for 'GENERATE' (CTRL + G), 'EXPLORE' (CTRL + SPACE), and a menu button.

Attention ! Choisissez la version de JAVA qui vous est disponible.

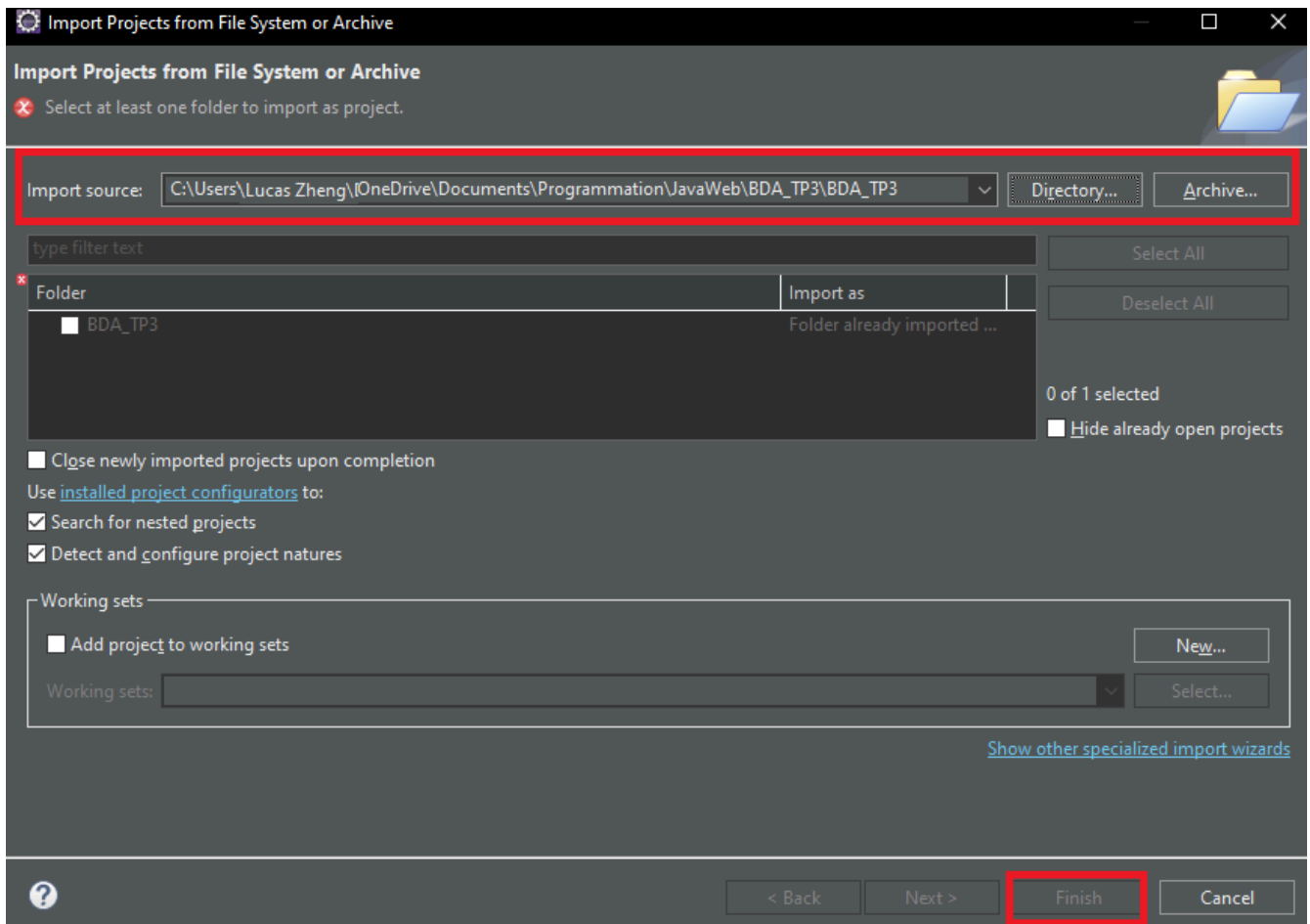
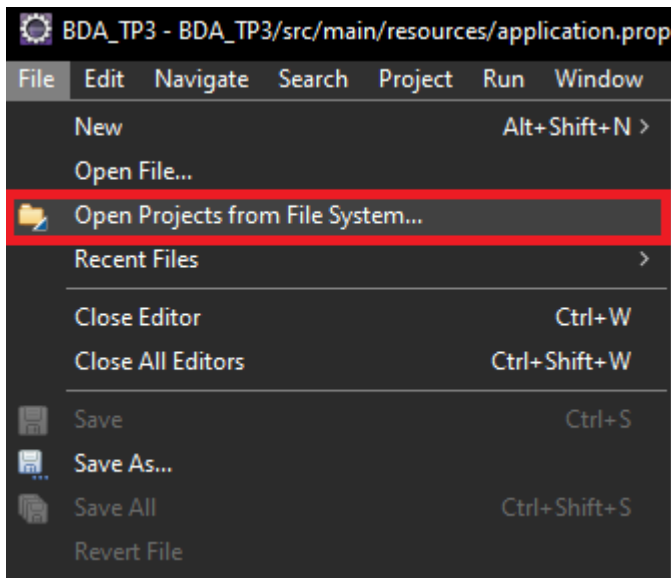
Dans mon cas, je possède la version 21 ( LTS ) avec la commande `java -version`

```
java version "21.0.7" 2025-04-15 LTS
Java(TM) SE Runtime Environment (build 21.0.7+8-LTS-245)
Java HotSpot(TM) 64-Bit Server VM (build 21.0.7+8-LTS-245, mixed mode, sharing)
```

Vous appuyez sur Generate pour télécharger le projet en zip puis l'extraire dans un répertoire de travail.

L'environnement de développement utilisé est Eclipse et je sélectionne comme workspace, le répertoire de travail choisi.

Enfin, ouvrez le projet en passant par File > Open Projects from File System... > Directory > Sélectionner le projet maven généré via le site Spring > Finish



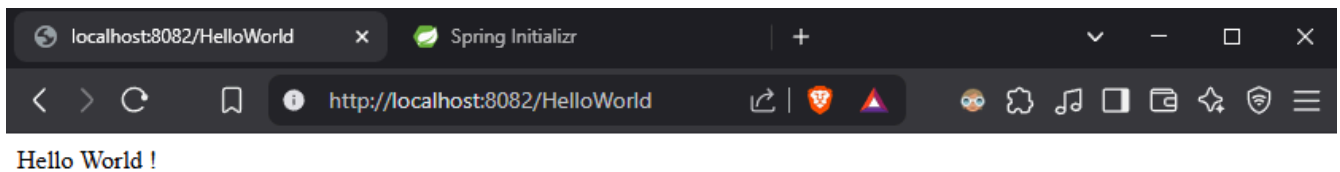
# Hello World

Ajoutez une nouvelle classe "My API" ( par exemple ) qui jouera le rôle de contrôleur dans l'architecture MVC

Le code est le suivant :

```
package com.example.BDA_TP3;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class MyAPI {
    @GetMapping("/HelloWorld")
    public String HelloWorld()
    {
        return "Hello World !";
    }
}
```

Enfin, lancez demoApplication.java ou BdaTp3Application.java puis entrez dans votre navigation web : "localhost:8080/HelloWorld"



Remarque :

La capture d'écran ci-dessus avait le port 8082. En effet, 8080 ( port par défaut pour Spring Boot ) était pris par un autre processus dans mon cas donc j'ai ajouté la ligne suivante dans le resources/application.properties :

```
server.port = 8082
```

# Spring Web

Munissez vous d'une application qui vous permet de tester les API notamment pour POST, PUT et DELETE afin de vérifier de votre côté les résultats de vos API.

Dans mon cas, j'utilise une extension google chrome dont le lien de téléchargement est ci-dessous :

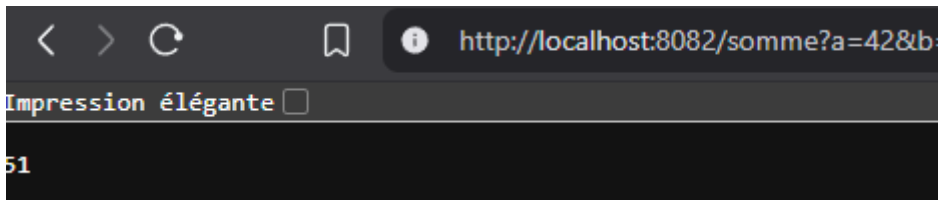
<https://chromewebstore.google.com/detail/talend-api-tester-free-ed/aejoelaoggembcahagimdiliamlcdmfm>

Vous pouvez maintenant enrichir le contrôleur MyAPI pour qu'elle fasse d'autres choses que Hello World comme faire la somme des deux entiers avec le code ci-dessous :

```
@GetMapping("/somme")
public int sommebis(@RequestParam int a,@RequestParam int b)
{
    return a+b;
}
```

Pour que vous mettez les paramètres, il faut faire la syntaxe suivante :  
/nom du mapping?paramètre1=valeur1&paramètre2=valeur2

Ainsi avec l'exemple de "<http://localhost:8080/somme?a=42&b=9>", on obtient



DRAFT

Save as

METHOD

GET

SCHEME :// HOST [ ":" PORT ] [ PATH [ "?" QUERY ]]

http://localhost:8082/somme?a=42&b=9

length: 36 byte(s)

Send

QUERY PARAMETERS

☒ a = 42

☒ b = 9

+ Add query parameter

HEADERS

Form

+ Add header

Add authorization

BODY

XHR does not allow payloads for GET request.

Response

Cache Detected - Elapsed Time: 96ms

200

HEADERS

pretty

Content-Type: application/json

Transfer-Encoding: chunked

Date: Thu, 24 Apr 2025 16:26:40 GMT

Keep-Alive: timeout=60

Connection: keep-alive

BODY

pretty

51

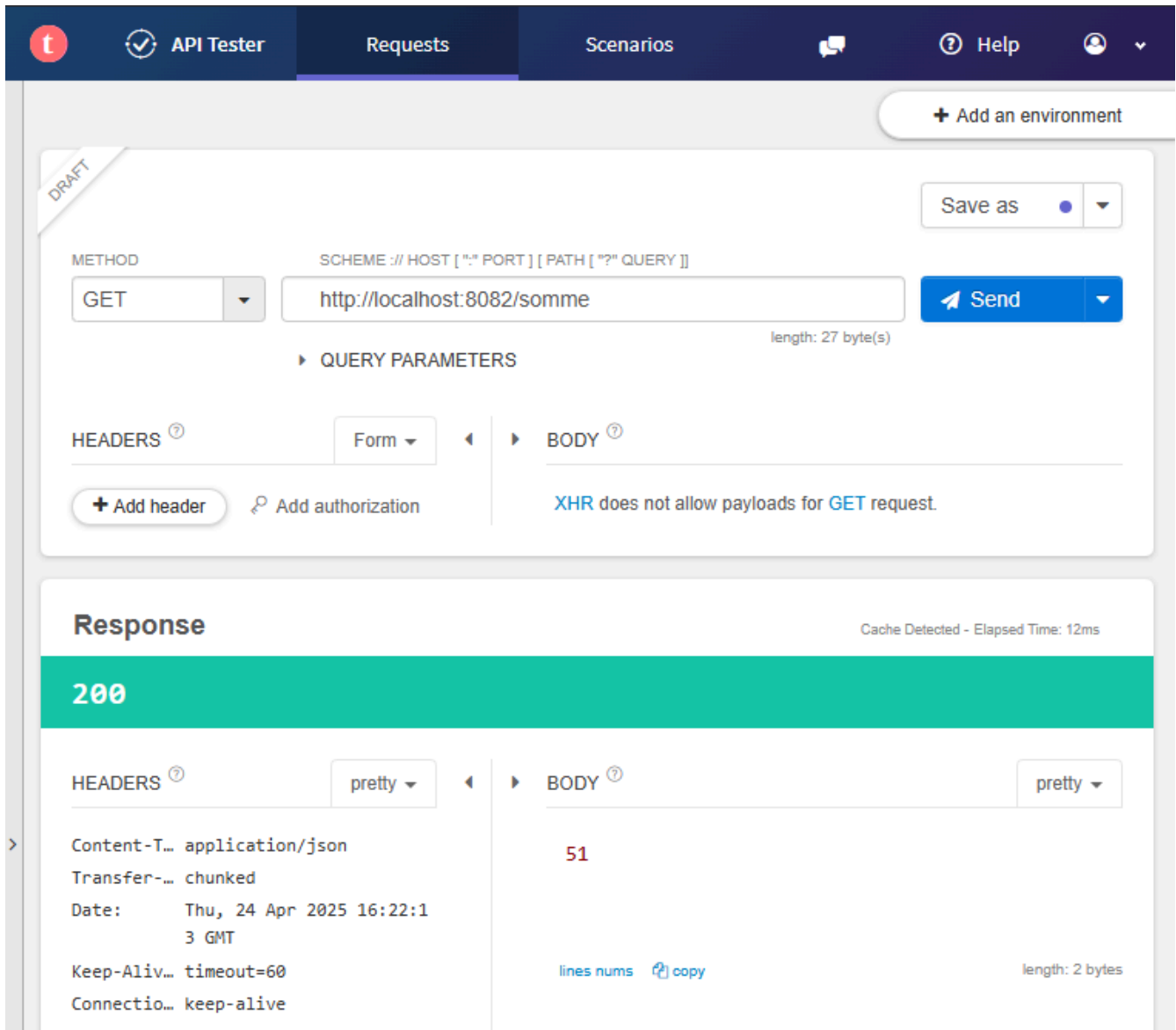
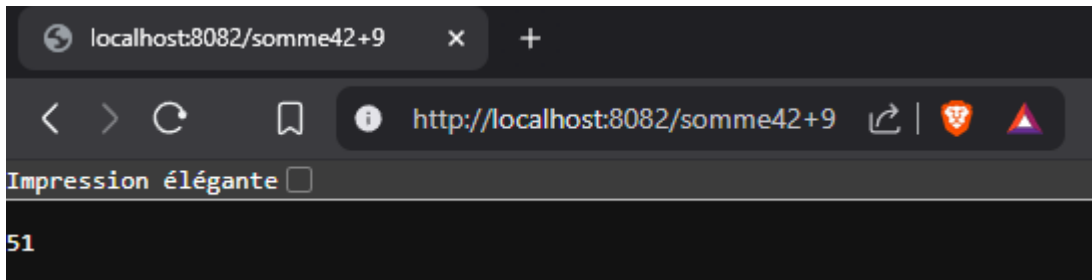
lines nums copy

length: 2 bytes

Si vous voulez éviter de se rappeler des noms de paramètre, vous pouvez passer directement avec PathVariable qui consiste à faire la substitution des chaînes de caractères en variable.

```
@GetMapping("/somme{a}+{b}")
public int somme(@PathVariable int a,@PathVariable int b)
{
    return a+b;
}
```

Ainsi avec l'exemple de "<http://localhost:8080/somme42+9>", on obtient

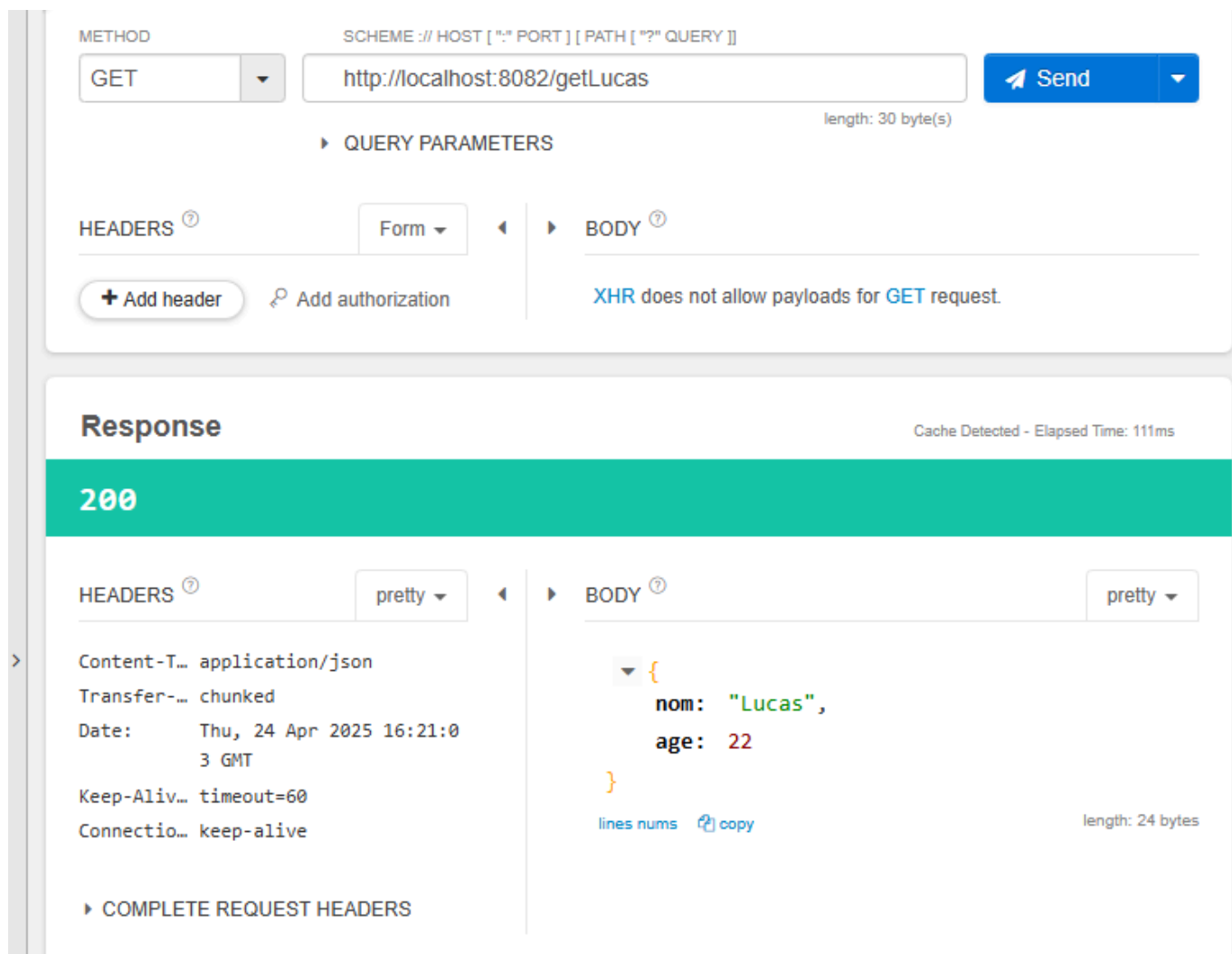
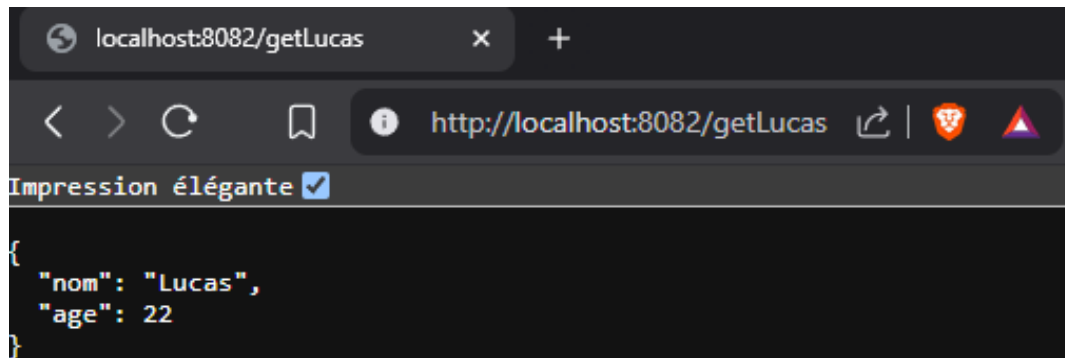




Vous pouvez aussi envoyer une classe comme client.

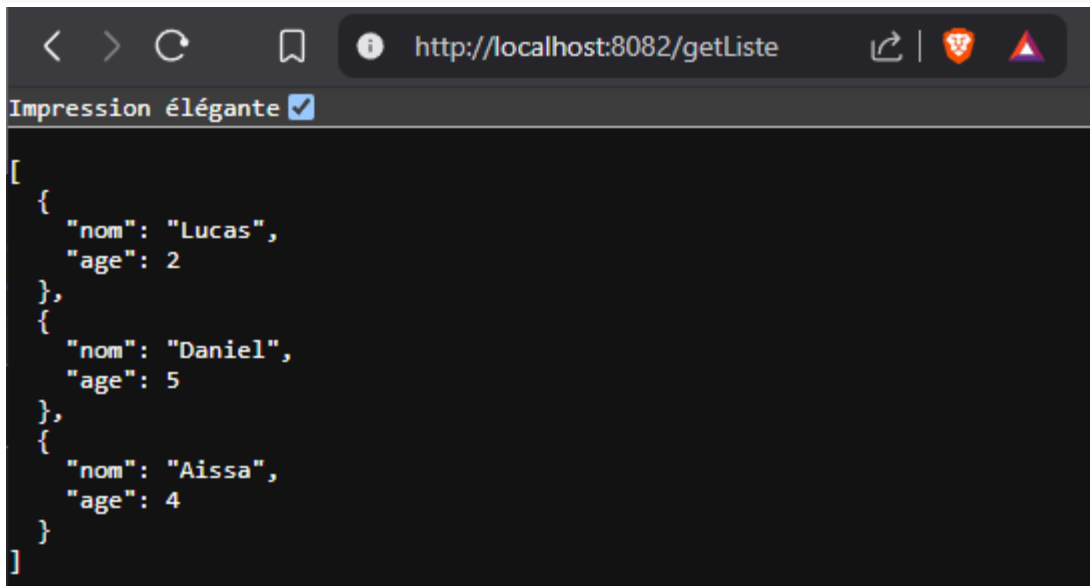
```
@GetMapping("/getLucas")
public ClientLocal getLucas()
{
    return new ClientLocal(1,"Lucas",22);
}
```

Ainsi, il envoie les données qui ont des getteurs comme nom et age mais pas id.



De plus, on peut aussi retourner une liste d'objets en format JSON avec le code suivant :

```
static
{
    ListeClients.add(new ClientLocal("Lucas",2));
    ListeClients.add(new ClientLocal("Daniel",5));
    ListeClients.add(new ClientLocal("Aissa",4));
}
@GetMapping("/getListe")
public ArrayList<ClientLocal> getListe()
{
    return ListeClients;
}
```



Ensuite, on peut ajouter un client dans une liste avec le code suivant :

```
@PostMapping("/addClient")
public String addClient(@RequestParam String nom, @RequestParam int age)
{
    ClientLocal client = new ClientLocal(nom,age);
    ListeClients.add(client);
    return "Client" + client +"ajouté !";
}
```

The screenshot displays the API Tester application interface. The top navigation bar includes a 't' icon, a checkmark icon, 'API Tester', 'Requests', 'Scenarios', a chat icon, a help icon, and a user profile icon. The 'Requests' tab is active.

**Request Configuration:**

- Method:** POST
- URL:** `http://localhost:8082/addClient?nom=Youcef&age=25`
- Length:** 49 byte(s)
- QUERY PARAMETERS:**
  - ☒ `nom` = Youcef
  - ☒ `age` = 25
  - [+ Add query parameter](#)
- HEADERS:**
  - ☒ `Content-Type` : application/json
  - [+ Add header](#)
  - [Add authorization](#)
- BODY:** Form (selected), Text (available). The body content is '1'.
- Footer:** Text JSON XML HTML | [Format body](#) | ☒ [Enable body evaluation](#) | [length: 0 byte](#)

**Response:**

Cache Detected - Elapsed Time: 34ms

**200**

**HEADERS:** pretty (selected).  
Content-Type: text/plain;charset=UTF-8  
Content-Length: 53 bytes

**BODY:** raw (selected).  
ClientClientLocal [id=3, nom=Youcef, age=25]ajouté !

Puis, modifier l'âge d'un client en connaissant son id :

```
@PutMapping("/putClient")
public String addClient(@RequestParam int id,@RequestParam int newage)
{
    ListeClients.get(id).setAge(newage);
    return "Client" + ListeClients.get(id) + " a été mis à jour !";
}
```

The screenshot displays the API Tester application interface. The top navigation bar includes a home icon, 'API Tester', 'Requests', 'Scenarios', a chat icon, 'Help', and a user profile icon. The 'Requests' tab is active.

**Request Configuration:**

- METHOD:** PUT
- URL:** http://localhost:8082/putClient?id=0&newage=22 (length: 48 byte(s))
- QUERY PARAMETERS:**
  - id = 0
  - newage = 22
- HEADERS:**
  - Content-Type: application/json
- BODY:** (Empty text area)

**Response:**

- Status:** 200 (Cache Detected - Elapsed Time: 5ms)
- HEADERS:**
  - Content-Type: text/plain;charset=UTF-8
  - Content-Length: 65 bytes
  - Date: Thu, 24 Apr 2025 16:35:44 GMT
- BODY:** ClientClientLocal [id=0, nom=Lucas, age=22] a été mis à jour !

At the bottom of the response body, there are buttons for 'Top', 'Bottom', '2Request', 'Copy', and 'Download'.

Enfin, supprimer un client en connaissant son id

```
@DeleteMapping("/deleteClient")
public String deleteClient(@RequestParam int id)
{
    ListeClients.remove(id);
    return "Client id " + id + " supprimé !";
}
```

METHOD: DELETE SCHEME: // HOST: [ ".:" PORT ] [ PATH [ "?" QUERY ] ]

http://localhost:8082/deleteClient?id=2 length: 39 byte(s)

QUERY PARAMETERS 1/2

☒ id = 2

+ Add query parameter

HEADERS 1/1 Form

+ Add header Add authorization

BODY 1/1

XHR does not allow payloads for DELETE request.

---

**Response** Cache Detected - Elapsed Time: 4ms

**200**

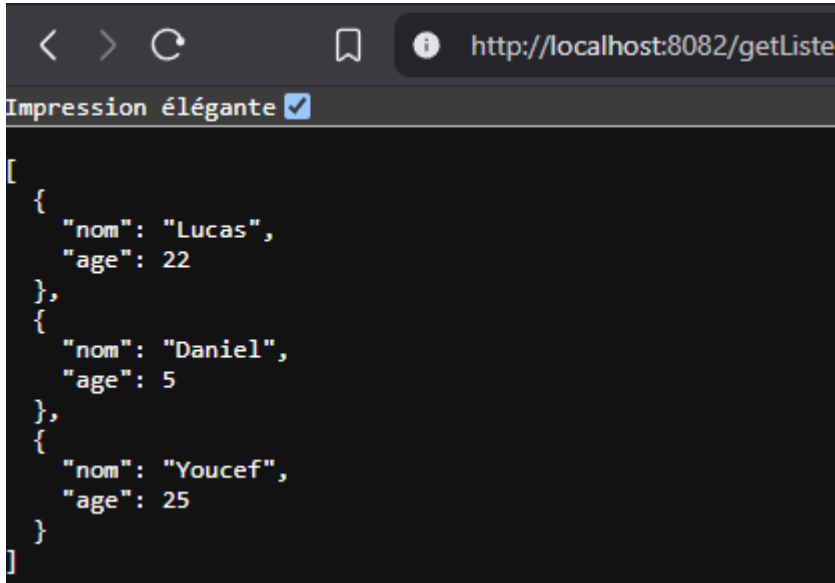
HEADERS 1/1 pretty

Content-Type: text/plain; charset=UTF-8

BODY 1/1 raw

Client id 2 supprimé !

Ainsi, après les opérations faites ci-dessus, nous obtenons une liste avec des données différentes



## H2 Database et JPA

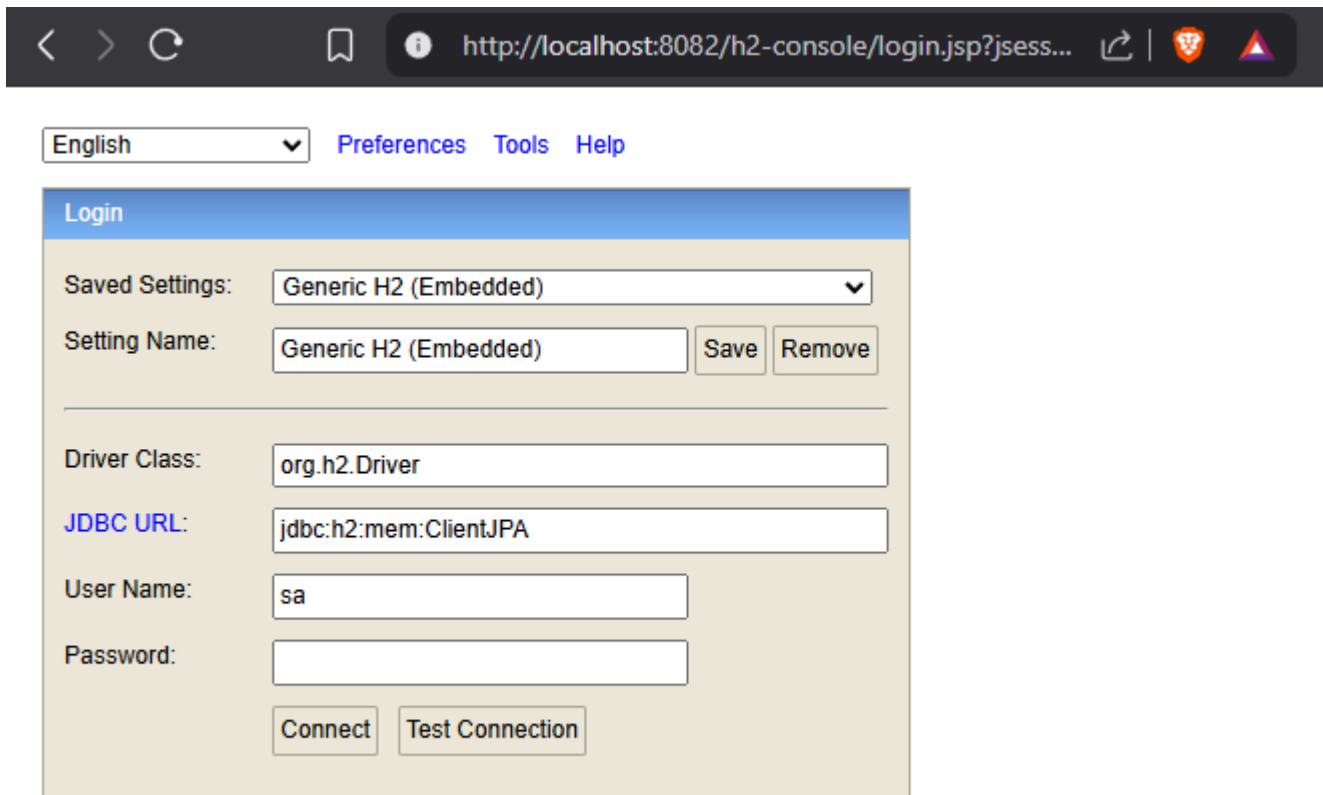
On modifie l'application pour qu'elle ajoute des données dans la base de donnée H2. Ainsi, la prochaine qu'on exécute l'application, on obtient toujours les trois clients

```
@SpringBootApplication
public class BdaTp3Application {
    public static void main(String[] args) {
        SpringApplication.run(BdaTp3Application.class, args);
    }
    @Bean
    CommandLineRunner runner(ClientJpaRepository repository)
    {
        return args ->
        {
            repository.save(new ClientJPA(0, "Uno", "Lucas", 2));
            repository.save(new ClientJPA(1, "Dos", "Daniel", 5));
            repository.save(new ClientJPA(2, "Tres", "Aissa", 4));
        };
    }
}
```

Puis on ajoute des lignes dans application.properties afin de pouvoir accéder à la console de la base de données H2 :

```
spring.datasource.url=jdbc:h2:mem:ClientJPA
spring.h2.console.enabled=true
```

Après avoir lancé le serveur, on entre dans le “localhost:8080/h2-console” qui va vous mener dans la page d’authentification ci-dessous :



The screenshot shows a web browser window with the address bar displaying `http://localhost:8082/h2-console/login.jsp?jsess...`. The page has a navigation bar with a language dropdown set to "English" and links for "Preferences", "Tools", and "Help". The main content area is titled "Login" and contains the following fields and buttons:

- Saved Settings:** A dropdown menu showing "Generic H2 (Embedded)".
- Setting Name:** A text input field containing "Generic H2 (Embedded)", with "Save" and "Remove" buttons to its right.
- Driver Class:** A text input field containing "org.h2.Driver".
- JDBC URL:** A text input field containing "jdbc:h2:mem:ClientJPA".
- User Name:** A text input field containing "sa".
- Password:** An empty text input field.
- Buttons:** "Connect" and "Test Connection" buttons at the bottom.

Vous pouvez appuyer sur Test Connection pour vérifier si le serveur détenant l'URL donné dans application.properties existe.

Test successful

Ainsi, vous pouvez appuyer sur “Connect” et vous arrivez à la page SQL dans laquelle vous pouvez gérer les données de repository comme si c’était une base de données SQL.

Ainsi, vous pouvez dans la capture ci-dessous, faire une sélection sur toute la table CLIENT;

The screenshot shows a SQL client interface with a toolbar at the top containing icons for undo, redo, auto-commit, max rows (set to 1000), and buttons for running queries. On the left, a tree view shows the database structure: jdbc:h2:mem:ClientJPA, CLIENT table, and its columns (ID, NBRPLACESRESERVEE, IDCLIENT, PRENOMCLIENT). Below the tree, it says 'H2 2.3.232 (2024-08-11)'. The main area has a 'SQL statement:' input field with the text 'SELECT \* FROM CLIENT;'. Below this, the results of the query are displayed in a table with 4 columns: ID, NBRPLACESRESERVEESCLEINT, IDCLIENT, and PRENOMCLIENT. The table contains 3 rows of data. Below the table, it says '(3 rows, 2 ms)' and there is an 'Edit' button.

ID	NBRPLACESRESERVEESCLEINT	IDCLIENT	PRENOMCLIENT
1	2	Uno	Lucas
2	5	Dos	Daniel
3	4	Tres	Aissa

Enfin, pour connecter directement le serveur à la base de donnée, il faut faire des modifications :

application.properties

```
spring.application.name=BDA_TP3
server.port = 8082 //optionnel
#spring.datasource.url=jdbc:h2:mem:ClientJPA
#spring.h2.console.enabled=true
spring.datasource.url=jdbc:postgresql://localhost:5432/TP5
spring.datasource.username=postgres
spring.datasource.password=Silhouette42
spring.jpa.hibernate.ddl-auto=create
```

pom.xml

```
</dependencies>
. . . . .
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
</dependencies>
```



Ainsi, on voit bien dans la base de donnée avec l'application de PostgreSQL ( pgAdmin 4 ) que la table client ait bien les données que nous avons ajoutées.

The screenshot shows the pgAdmin 4 web interface. The top menu bar includes File, Object, Tools, Edit, View, Window, and Help. The main toolbar contains various icons for database management. The left sidebar shows the database structure tree. The central pane displays a SQL query: `select * from client;`. Below the query, the 'Data Output' tab is active, showing the results of the query in a table format. The table has four columns: `id` (integer, primary key), `nbplacesreserveesclient` (integer), `idclient` (character varying (255)), and `prenomclient` (character varying (255)). The results show three rows of data.

	id [PK] integer	nbplacesreserveesclient integer	idclient character varying (255)	prenomclient character varying (255)
1	0	2	Uno	Lucas
2	1	5	Dos	Daniel
3	2	4	Tres	Aissa