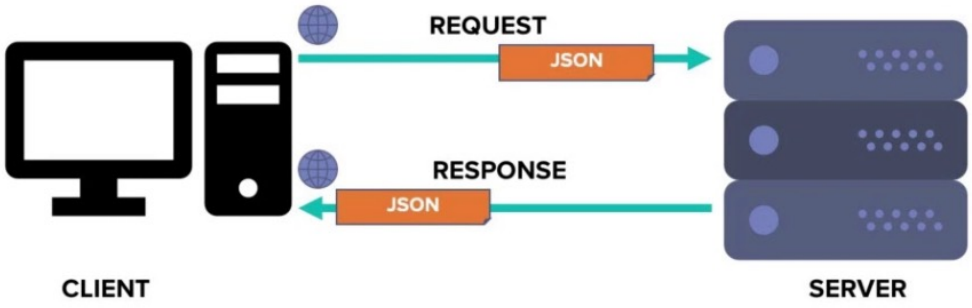
Coding

EzWh implementation

EzWh

- EzWh is a web application composed of:
  - A web server, which exposes HTTP/Rest/JSON APIs
  - A web client which interacts with the server through HTTP/Rest/JSON calls

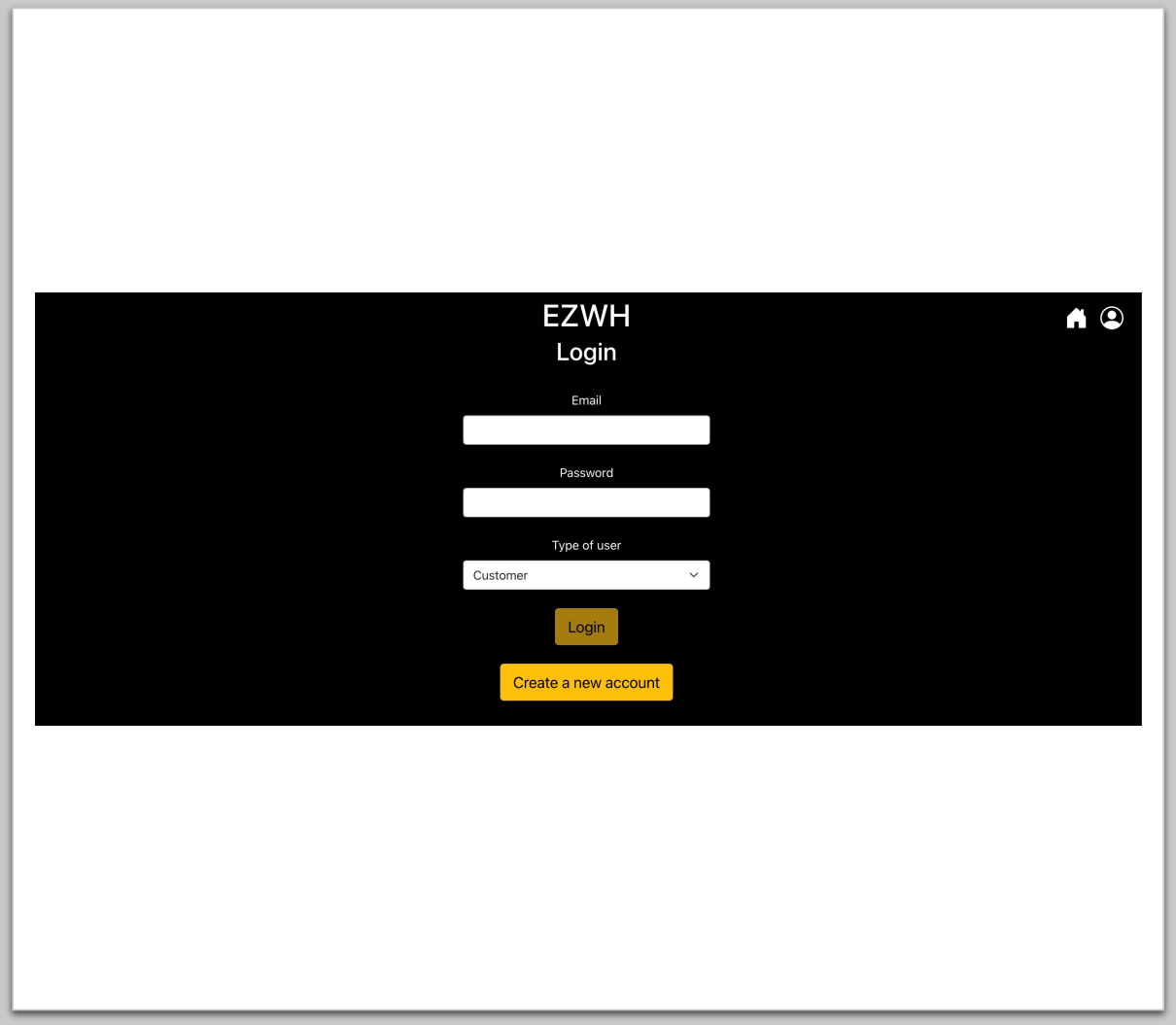


EzWh Client

- Already implemented in React
- Will be available in your repo in the `code/client` folder
- Uses the APIs defined in API.md
- Implements the use case/scenarios defined in `OfficialRequirements.md`
- In the current state it will not work because the server is not (yet) implemented

EzWh Client

- Login page for the frontend



EzWh Server

- Sketched in node.js (with the Express module)
- (Soon) Available in your repo in the `code/server` folder
- Will provide the implementation of the HTTP/Rest/Json APIs defined in `API.md`
- In the current state it will not work because you need to provide the implementation based on the requirements defined in the `OfficialRequirements.md` and `API.md`

EzWh Server

- Server source code

```
'use strict';
const express = require('express');
// init express
const app = new express();
const port = 3001;

app.use(express.json());

//GET /api/test
app.get('/api/hello', (req,res)=>{
  let message = {
    message: 'Hello World!'
  }
  return res.status(200).json(message);
});

// activate the server
app.listen(port, () => {
  console.log(`Server listening at http://localhost:${port}`);
});

module.exports = app;
```

npm package manager

- EzWh client and server use the npm package manager
- npm is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment Node.js
- It consists of a command line client, also called npm, and an online database of public and paid-for private packages, called the npm registry
- If you do not have it yet, please install npm in your local machine!

npm package manager

- When using the npm package manager, two files will be created
  - `package.json`: holds various metadata relevant to the project. It gives information to npm to handle the project's dependencies and some shortucts to execute complex scripts (e.g., for testing the code). More info at https://docs.npmjs.com/cli/v6/configuring-npm/package-json
  - `package-lock.json`: keeps track of the exact version of every package that is installed

EzWh client package.json

- The `dependencies` section allows the `npm install` command to install all the required dependencies
- The `scripts` section allows the `npm start` command to run the script called `react-scripts start`
- Other sections contain metadata

```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.16.1",
    "@testing-library/react": "^12.1.2",
    "@testing-library/user-event": "^13.5.0"
    "bcryptjs": "^2.4.3",
    "bootstrap": "^5.1.3",
    "react": "^17.0.2",
    "react-bootstrap": "^2.1.2",
    "react-bootstrap-icons": "^1.7.2",
    "react-dom": "^17.0.2",
    "react-router": "^6.2.1",
    "react-router-dom": "^5.2.0",
    "react-scripts": "^5.0.0",
    "react-toastify": "^8.1.1",
    "web-vitals": "^2.1.3"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "proxy": "http://localhost:3001",
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

EzWh server package.json

- The `dependencies` section allows the `npm install` command to install all the `express` dependency
- The `dev-dependencies` section allows the `npm install` command to install all the testing dependencies which will be use for development only
- The `scripts` section allows the `npm test` command to run the unit tests, and the `npm run apiTest` to run the integration tests

```
{
  "dependencies": {
    "express": "^4.17.3"
  },
  "devDependencies": {
    "babel": "^6.23.0",
    "chai": "^4.3.6",
    "chai-http": "^4.3.0",
    "jest": "^27.5.1",
    "mocha": "^9.2.2"
  },
  "scripts": {
    "apiTest": "./node_modules/.bin/mocha test --exit",
    "test": "node_modules/.bin/jest"
  }
}
```

Running EzWh client

- Open a terminal
- Open the `code/client` folder
- Type `npm install`
- Type `npm start`

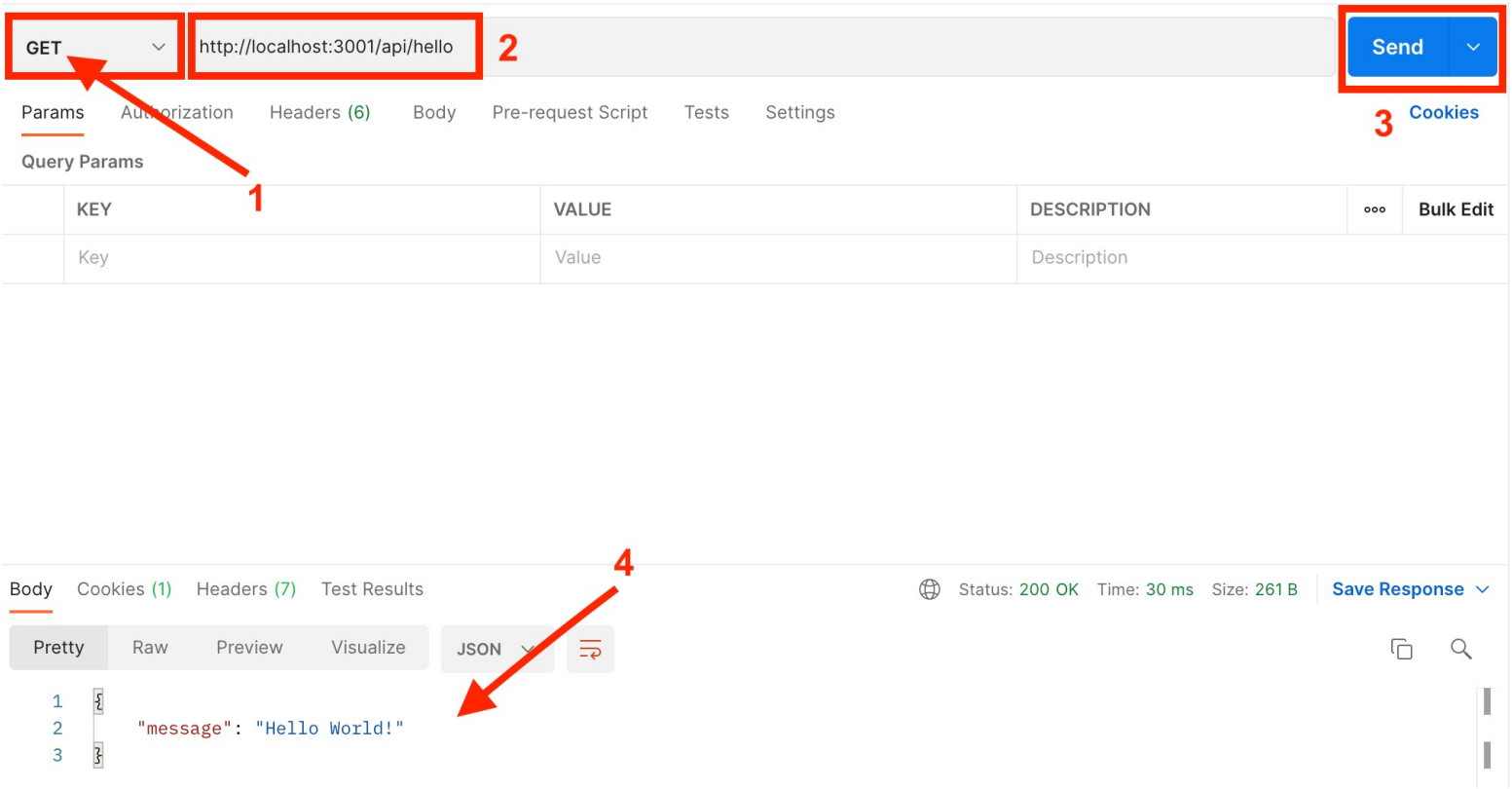
```
Compiled successfully!

You can now view client in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.8.18:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

- A browser window will be opened in http://localhost:3000 with the login page showed in slide 5.
- **Do not close the terminal window**, otherwise the client will be shutted down

Running EzWh server

- Open a terminal
- Open the `code/server` folder
- Type `npm install`
- Type `node server.js`
- This message will be shown: `Server listening at http://localhost:3001`
- **<u>Do not close the terminal window</u>**, otherwise the server will be shut down

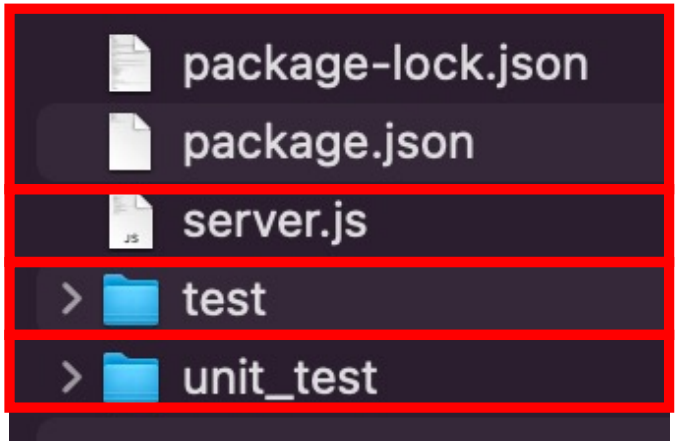
How to try the EzWh server (with Postman)



Server folder structure

Files

- In your repo you will find those files:



npm configuration files

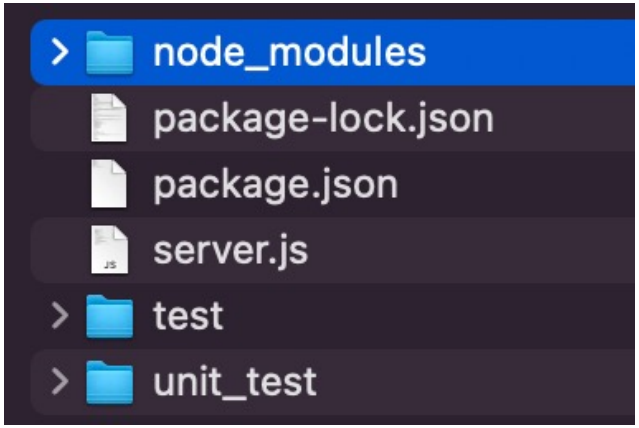
EzWh server source

Folder for the integration tests

Folder for the unit tests

- It is possible to add folders and files, **<u>do not</u>** move these files!

After Running npm install



Installed dependencies

How to add an API to the EzWh server

- The server will manage HTTP request, so it needs to manage
  - GET requests
  - POST requests
  - PUT requests
  - DELETE requests
- Thanks to the `express` module, this is extremely simple.

```
//GET /api/test
app.get('/api/hello', (req,res)=>{
  let message = {
    message: 'Hello World!'
  }
  return res.status(200).json(message);
});
```

Adding a POST API

- As a playground example, we want to add an HTTP/POST API to the server that sorts an array received in the request body as a JSON file
- We need to
  1. Implement a module that implements a sorting algorithm (e.g., a Bubblesort)
  2. Create an HTTP POST api
  3. Get the array from the JSON in the request body
  4. Include the module in the server in the function that implements the API
  5. Call the function with the array
  6. Send back the sorted array as a JSON in the POST API response

Creating a new js module

- Create a new file (`Sort.js`) and add it in the server folder
- Create for example a new class Sort and add the bubblesort function inside it
- Export the class Sort. In this way it will possible to import it inside the server source code.

```javascript
class Sort {
  constructor() {}

  bubblesort(array) {
    let isSorted = false;
    while (!isSorted) {
      isSorted = true;
      for (let i = 0; i < array.length - 1; i++) {
        if (array[i] > array[i + 1]) {
          [array[i], array[i + 1]] = [array[i + 1], array[i]];
          isSorted = false;
        }
      }
    }
    return array;
  }
}
module.exports = Sort;
```

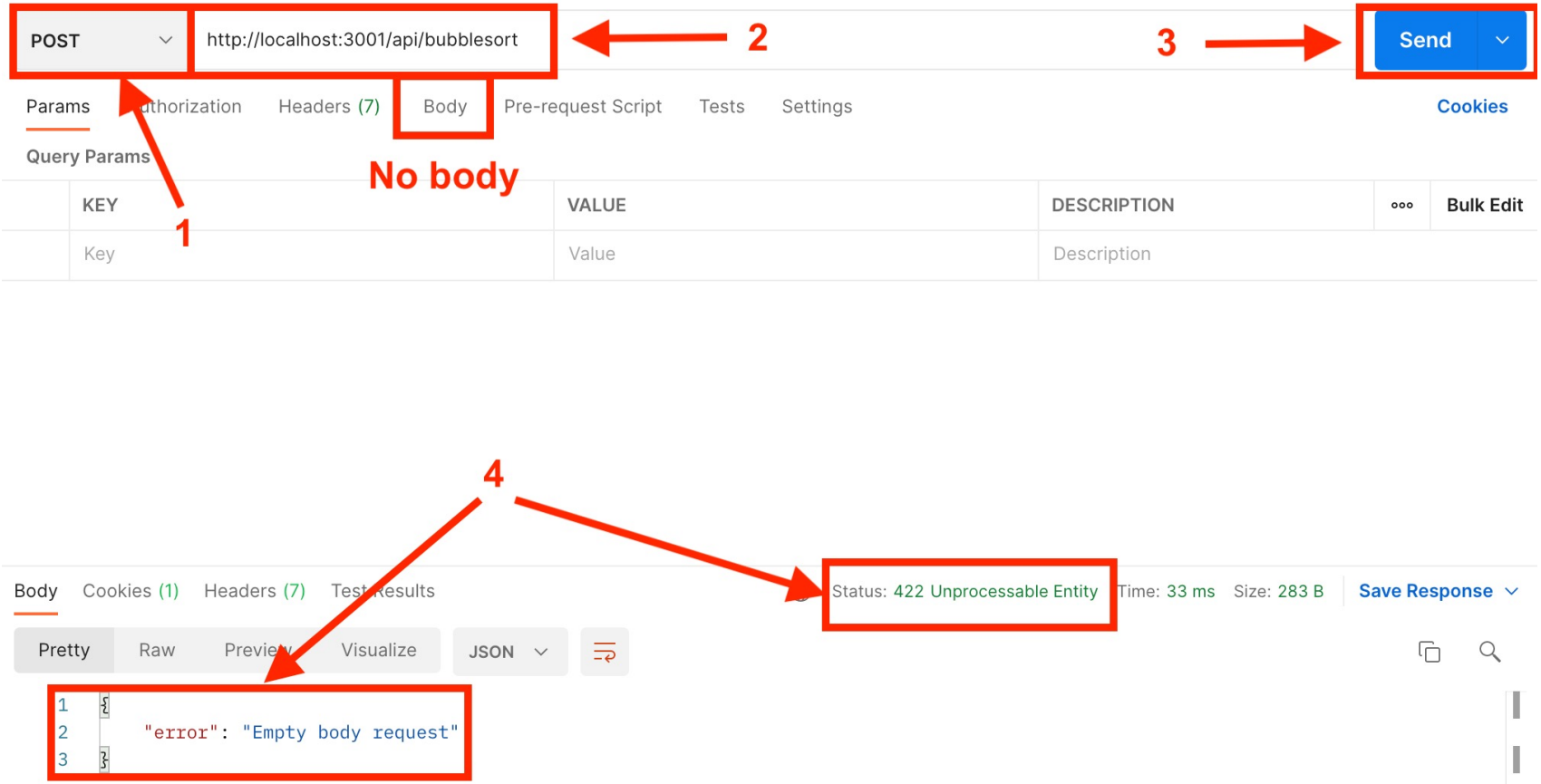
Creating a new REST API

- Add a new app.post as showed in the example.
- This function will manage the http request in the req parameter and the http response in the res parameter.
- In this example the function will return an error message if there is no json file in the http request with error code 422, and it will return a 200 code if it finds anything in the http request.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

```
app.post('/api/bubblesort', (req, res) => {
  //Check input parameters
  if (Object.keys(req.body).length === 0) {
    return res.status(422).json({error: `Empty body request`});
  }
  return res.status(200)
});
```

API Call with POSTMAN

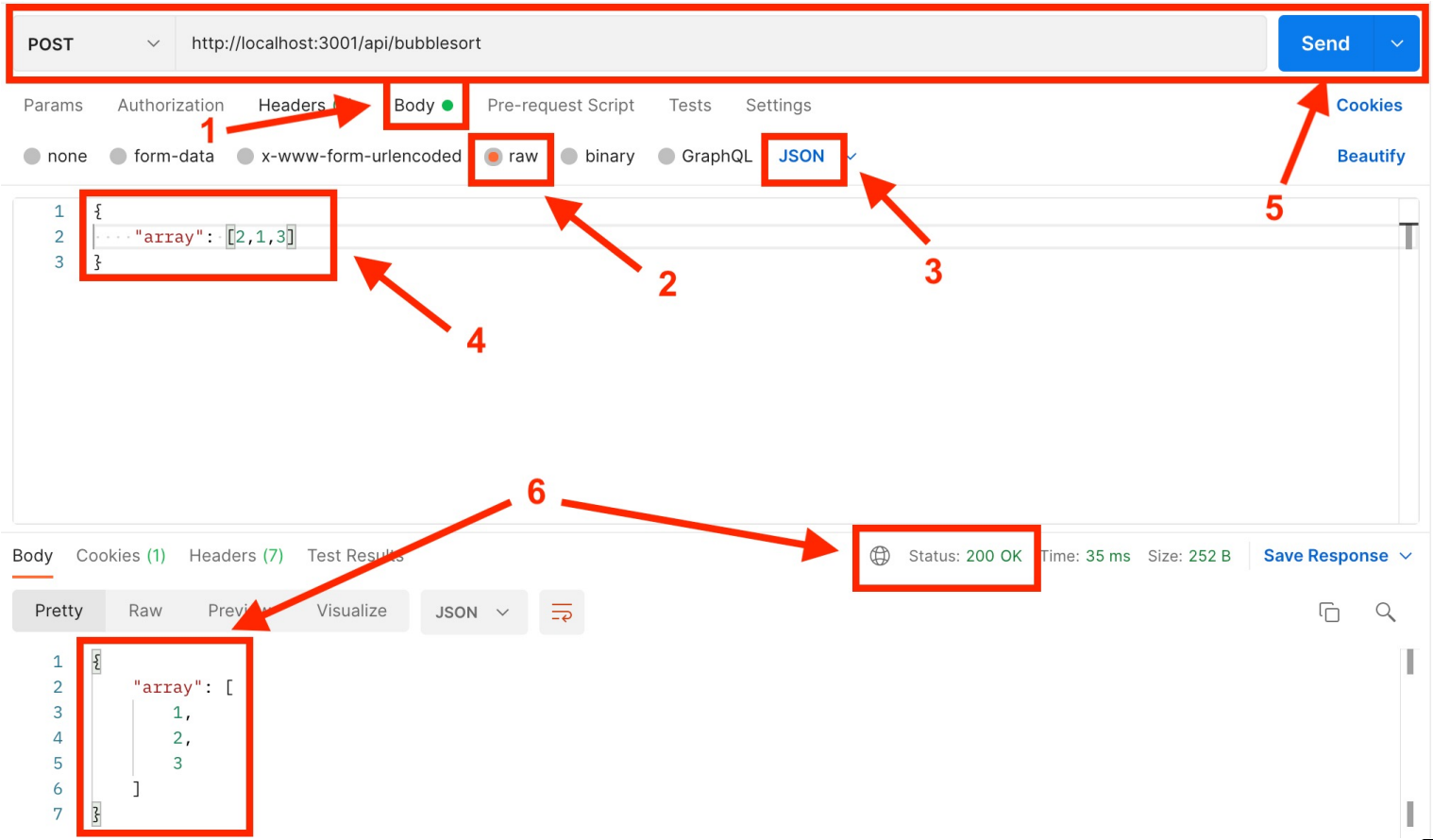


Adding the Sort Class

- First we need to import the `Sort` module. It's important to export it in the class source file, otherwise it will not work
- Express transforms the JSON file in the HTTP request in an object. `{"array": [2,1,3] }` becomes `req.body.array`
- `.json` function transforms a js object in a json file sent as an HTTP response.

```
const Sort = require('./Sort');

app.post('/api/bubblesort', (req, res) => {
  //Check input parameters
  if (Object.keys(req.body).length === 0) {
    return res.status(422).json({error: `Empty body request`});
  }
  let arrayToSort = req.body.array;
  let sortedArray = new Sort().bubblesort(req.body.array);
  let result = {array: sortedArray}
  return res.status(200).json(result);
});
```

API Call with Postman



Persistence

Persistence

- It is possible to choose how to implement the persistence
  - SQLite
  - H2
  - Files (please don't!)
  - …
- **Do not use** MYSQL, Oracle DB, or others that need a machine configuration.
- Only use DB that stores data in a file inside the `code/server` folder
- **DO NOT STORE PASSWORDS IN PLAIN TEXT!!!**

Persistence with sqlite3

- `npm install sqlite3`
- Create a module for interacting with the db
- Create Unit Tests
- Use the module in the server

Persistence module

- **The constructor of this playground example creates the new db (if does not exist) with the name passed as a parameter**
- **Deletes the NAMES table**
- **Creates a new NAMES table**
- Adds a new user (with name and surname)
- Gets All the users

```javascript
class DAO {
    sqlite = require('sqlite3');
    constructor(dbname) {
        this.db = new this.sqlite.Database(dbname, (err) => {
            if(err) throw err;
        });

    }
    dropTable() {
        return new Promise((resolve, reject)  => {
            const sql = 'DROP TABLE IF EXISTS NAMES';
            this.db.run(sql, (err) => {
                if (err) {
                    reject(err);
                    return;
                }
                resolve(this.lastID);
            });
        });
    }
    newTableName() {
        return new Promise((resolve, reject)  => {
            const sql = 'CREATE TABLE IF NOT EXISTS NAMES(ID INTEGER PRIMARY KEY AUTOINCREMENT,NAME VARCHAR, SURNAME VARCHAR)';
            this.db.run(sql, (err) => {
                if (err) {
                    reject(err);
                    return;
                }
                resolve(this.lastID);
            });
        });
    }
```

Persistence module

- The constructor of this playground example creates the new db (if does not exist) with the name passed as a parameter
- Deletes the NAMES table
- Creates a new NAMES table
- **Adds a new user (with name and surname)**
- **Gets All the users**

```
33      storeUser(data) {
34          return new Promise((resolve, reject) => {
35              const sql = 'INSERT INTO NAMES(NAME, SURNAME) VALUES(?, ?)';
36              this.db.run(sql, [data.name, data.surname], (err) => {
37                  if (err) {
38                    reject(err);
39                    return;
40                  }
41                  resolve(this.lastID);
42              });
43          });
44      }
45      getStoredUsers() {
46          return new Promise((resolve, reject) => {
47              const sql = 'SELECT * FROM NAMES';
48              this.db.all(sql, [], (err, rows) => {
49                  if (err) {
50                      reject(err);
51                      return;
52                  }
53                  const names = rows.map((r) => (
54                      {
55                          id:r.ID,
56                          name : r.NAME,
57                          surname : r.SURNAME
58                      }
59                  ));
60                  resolve(names);
61              });
62          });
63      }
64  }
65
66  module.exports = DAO;
```

Express Application Generator

It is possible to generate the server project by using the Express Application Generator:
https://expressjs.com/en/starter/generator.html

```
npm install –g express-generator
express –no-view --git
```

This tool will create a project with a proper structure

Example Repository

The code presented in these slides is available at this link:

- Starting code: https://git-softeng.polito.it/d023270/EzWh (master branch)
- Presented code: https://git-softeng.polito.it/d023270/EzWh (playground branch)