Programmer's guide

Table of contents

Foreword

This document is intended for developers who want to understand the codebase of the application and contribute to its development. It provides an overview of the code structure, how to set up the development environment, and how to deploy the application. It also includes a deep dive into the crucial parts of the codebase, including Svelte files, Python code, and the use of Tailwind and DaisyUI.

If you are looking for a more general overview of the application, please refer to the User's guide.

Getting started

To get started with the application, you need to set up your development environment. This includes installing the necessary dependencies, configuring the application, and running it locally.

Installation

To install the application, you first need to clone the repository. You can do this by running the following command in your terminal:

```
git clone https://github.com/FloSto-Bash/Bac3-Projet-Individuel
cd Bac3-Projet-Individuel
```

Then, you need to install the required dependencies. The application uses npm for JavaScript dependencies. To install the JavaScript dependencies, run the following command in the root directory of the project:

```
npm install
```

This will install all the necessary JavaScript packages listed in the `package.json` file.

Now, everything is set up!

Running the application

To run the application, you need to start the development server. You can do this by running the following command in the root directory of the project:

```
cd project
npm run dev
```

This will start the development server. The application will be available at `http://localhost:5173`.

You can now browse the application and start testing its features.

Testing

To run the tests, you need to have vitest and pytest installed. You can install them by running the following command in the root directory of the project:

```
npm run test
```

This will run the JavaScript tests using Vitest. The tests are located in the `src/lib/tests` directory.

To run the Python tests, you can run the following command **in the root directory of the project**:

```
cd ..
python3 -m unittest project/static/python/tests/test_restricted_checks.py
```

This will run the Python tests using unittest. The test is located in the `project/static/python/tests` directory.

Some codes are not tested. The reasons are explain in a justification file.

Code structure

```
project/
|---- ...
```

```
|---- .svelte-kit/
|   |---- ...
|---- src/
|   |---- app.css
|   |---- app.html
|   |---- lib/
|   |   |---- ...
|   |---- routes/
|   |   |---- ...
|   |   |---- +page.svelte
|   |   ----- components/
|   |       |---- diagram.svelte
|   |       ----- editor.svelte
|---- static/
|   |---- icon/
|   |---- logo/
|   |---- python/
|   |   |---- src/
|   |   |   |---- main.py
|   |   |   |---- restricted_checks.py
|   |   |   |---- worker.py
|   |   ----- config/
|   |       ----- pyscript.json
```

Where to find...

In this section, you will find the location of the project's main files.

Source code

The project is divided into two main parts:

1. The **front-end**, which is built using Svelte and Tailwind CSS. It is located in the `src` directory.
   - app.css: The main CSS file for the application.
   - app.html: The main HTML file for the application.
   - `src/lib/`: Contains the main Svelte components and utilities. The files located in this directory are used directly in the Svelte components.
     - extractIntegers.js: A JavaScript utility function used to extract integers from a string.
     - initialCode.js: Store the initial code, which is displayed in the editor when the application is first loaded.
     - selectedList.js: Stored the selected list, which is chosen by the user and displayed in the diagram.
     - unsortedList.js: Store the unsorted list, which are the different lists available in the application.
   - `src/routes/`: Contains the Svelte routes and components.
     - +page.svelte: The main page of the application.

    - `src/routes/components/`: Contains the Svelte components used in the application.
        - diagram.svelte: The component used to display the diagram.
        - editor.svelte: The component used to edit the diagram.
2. The **client-side Python code**, which is used to run the Python scripts and communicate with the front-end. Located in the `static/python`, this part of the project aims to execute Python code in the browser, using Pyscript.
    - `static/python/config/`: Contains the configuration files for Pyscript. You will find more details about the configuration files in the Deep dive section section.
        - pyscript.json: The configuration file for Pyscript. It contains the settings for the Python environment and the modules to be loaded.
    - `static/python/src/`: Contains the Python source code files.
        - main.py: The main Python file that is executed when the 'execute' button is clicked. It contains the main logic of the Python code.
        - worker.py: The worker file that is used to run the Python code in a separate thread. It is used to avoid blocking the main thread and to allow the application to remain responsive while the Python code is running.
        - restricted_checks.py: The file that contains the restricted checks for the Python code. It is used to ensure that the Python code does not execute any dangerous or restricted operations.

Documentation

The documentation is located in the `doc` directory. It contains the following files:

1. The source code of the documentation, which is written in Markdown, is in `doc/sources`. The documentations are transformed into LaTeX files and then into PDF files using pandoc.
2. The generated documentation, which is in PDF format, is in `doc/PDFs`. The PDF files are generated using Pandoc. However, I recommend using the source code of the documentation, with a Markdown interpreter to read it. Actually, as the documentation is written with multiple links, it is easier to read it directly in a markdown interpreter, such as in GitHub.

Documents are available for users and programmers.

Development environment

In order to develop the application, I used different tools and libraries. This section describes the development environment I used to develop this application, so you can reproduce it on your own machine.

Languages

The application is built using the following languages:

- **JavaScript**: The front-end is built using JavaScript. The application uses Svelte as the main framework for building the user interface.
- **Python**: The client-side is built using Python. The application uses Pyscript to run Python code in the browser.

- **HTML**: The application uses HTML to structure the web pages.
- **CSS**: The application uses CSS for styling. The application uses Tailwind CSS for utility-first CSS styling and DaisyUI for component-based styling.
- **Markdown**: The documentation is written in Markdown, a lightweight markup language for formatting text. The documentation is generated using pandoc, a universal document converter.

Frameworks

The application uses the following frameworks:

- **Svelte**: The front-end is built using Svelte, a modern JavaScript framework for building user interfaces. Svelte is a component-based framework that allows you to build reusable components and manage state easily.
- **Tailwind CSS**: The application uses Tailwind CSS for utility-first CSS styling. Tailwind CSS is a utility-first CSS framework that allows you to build custom designs without leaving your HTML.
- **DaisyUI**: The application uses DaisyUI for component-based styling. DaisyUI is a Tailwind CSS component library that provides pre-designed components that you can use in your application.
- **d3-scale**: The application uses d3-scale for scaling and mapping data. D3 is a JavaScript library for producing dynamic, interactive data visualizations in web browsers. The d3-scale module provides the function `scaleLinear`, which is used to create a linear scale for mapping data values to visual values.
- **Pyscript**: The application uses Pyscript to run Python code in the browser. Pyscript is a framework that allows you to run Python code in the browser using Pyodide, a Python interpreter for WebAssembly.
- **AST**: The application uses the Abstract Syntax Tree (AST) to parse and analyze Python code. The AST is a representation of the structure of the Python code, which allows you to analyze and manipulate the code easily.
- **Vitest**: The application uses Vitest to test the JavaScript code. Vitest is a fast unit test framework powered by Vite.
- **Unittest**: The application uses Unittest for testing the Python code. Unittest is a built-in Python module for testing Python code.

How to deploy

To deploy the application on a server, you will need to follow these steps:

1. Clone the repository to your server using the following command:

```
git clone https://github.com/FloSto-Bash/Bac3-Projet-Individuel
```

2. Install the required dependencies using npm:

```
npm install
```

3. Build the application using the following command:

```
npm run build && npm run package
```

4. Copy the `dist` directory to your server's web root directory. The `dist` directory contains the built application files.

5. Configure your web server to include the following HTTP headers :

```
'Access-Control-Allow-Origin'  : '*'
'Cross-Origin-Opener-Policy': 'same-origin'
'Cross-Origin-Embedder-Policy': 'require-corp'
'Cross-Origin-Resource-Policy': 'cross-origin'
```

Deep dive

Here, some details will be given about the most important files of the project. You must understand these files to be able to modify the application. However, the code is already well documented, so you can read the code and understand it. You will only find an overview of the files and some technical details here.

Svelte files

The Svelte files are divided into three main files :

+pages.svelte The code in this file aims to create the main layout of the application. It imports the `Diagram` and `Editor` components, which are used to display the components of the application.

It also initializes the `selectedList` and `unsortedList` variables. The `selectedList` variable is initialized with the first list in the `unsortedList` array. Moreover, it initializes the subscription to the user's code and statistics. The subscription is used to store the user's code and statistics in the local storage, every 2 seconds.

Finally, it contains the logic to import and export the user's code, using file input and output.

To conclude, this file is the main entry point of the application. It contains the main layout of the application and imports the other components.

diagram.svelte This file contains the code to display the diagram. It uses the `d3-scale` library to create a linear scale for mapping data values to visual values.

This component allows the user to do the following actions: - Select a list to sort from the `unsortedList` array. - Activate/deactivate the compare and swap functionalities. - Display the diagram of the selected list, or the animation of the sorting algorithm. - Reset the statistics of the sorting algorithm, launch the algorithm, and reset the diagram to its shuffled state. - Display the statistics of the sorting algorithm, such as the number of comparisons and swaps.

All of these functionalities are implemented in this section.

As said before, please refer to the code to understand how it works. However, I will give you some details concerning the most important part of this file: the `onMount` function. This function is called when the component is mounted. It is used to assign all the important variables and functions to the window object. This allows us to access these variables and functions from the Python code, nice, right?

editor.svelte This file contains the code to display the editor. It uses the `Monaco Editor` library to create a code editor in the browser. The Monaco Editor is a powerful code editor that supports syntax highlighting, code completion, and other features. You might already know it, as it is used in Visual Studio Code.

In this file, functions are defined to handle the following actions: - Initialize the editor with the user's code. - Update the user's code when the editor content changes. - Update the editor dimensions when the window is resized.

Python Code

Now, let's take a look at the Python code.

First, you need to understand the configuration of a Pyscript application. The Pyscript environment is configured using a JSON file, which is located in the `static/python/config` directory. In the JSON file, the following options are defined:

1. Package: mentions the packages to be loaded in the Pyscript environment.
2. Interpreter: Oblige the Pyscript interpreter to use a specified version of Pyodide. The reason for this is explained in the justification file.
3. Files: This option is used to load the Python files in the PyScript environment. This allows importing the Python files in the Pyscript environment and using them in the application. Sadly, this does not work at the moment, so the files are copied where it is needed. I have no explanation for this.

To know more about Pyscript configuration, please refer to the Pyscript documentation.

The Python code is also divided into three files:

main.py This file contains the main logic of the Python code.

First, you can see this file as the executed Python code on the main thread. The first function defined in this file is called from the front-end, when the user clicks on the 'execute' button. Then, this function is used to :

1. Create two workers, one for the time execution and one for the animation.
2. Lunch the two workers, starting with the time execution worker.
3. Waiting for the timed execution worker to finish.
4. Handle the result of the time execution worker, raising exceptions if needed.
5. Do the same for the animation worker, by waiting for the animation worker to finish and handling the result of the animation worker.
6. Finally, it terminates the two workers.

You might wonder why the two workers are created and terminated at each execution. The reason is that I did not find a way to create two workers in the `onMount` function from the Svelte file. This would have been more efficient and more elegant.

The communication between the main thread and the workers is done using the `PyWorker` class from Pyscript. In addition to that, the separate workers can 'send messages' to the main thread using the `postMessage` method. This allows sending data between the main thread and the workers. It is used to handle errors, as raising an error in a separate thread would not affect the main thread.

After that, notice that multiple functions are defined at the end of the file. These functions are used to handle the communication between the Python code and the front-end.

worker.py The user's code is executed in this file. It contains the logic to execute the user's code, communicate with the main thread, and a duplicate of the `restricted_checks.py` file.

First, notice the following two crucial functions: `compare` and `swap`. These functions are used to compare and swap the elements of the list. The user can use these functions in their code to compare and swap the elements of the list. These functions ensure the link between the User's code and the front-end. They must be used in the user's code to see an animation of the sorting algorithm.

Then, you might notice the `define_global` function. This function defines the global environment for the execution of the user's code. It creates a global environment that contains the `compare` and `swap` functions, as well as the current `selectedList` and a set of predefined modules. To import other modules, the user can use the `import` statement in their code. However, please refer to the restricted_checks.py section to know which modules are available for the user.

Finally, the `execute` function is used to execute the user's code. It is this function that is called from the 'execute' button in the front-end. This function does the following actions:

1. Get the global environment for the execution, from the `define_global` function.
2. Parse the user's code using the `ast` module. The `ast` module is used to parse the Python code and create an Abstract Syntax Tree (AST) representation of the code. This ensures that the user's code is safe for execution.
3. Compile the user's code and execute the user's code
4. Return the sorted list to the main thread.

restricted_checks.py The `restricted_checks.py` file contains the restricted checks for the Python user's code. It is used to ensure that the Python code does not execute any dangerous or restricted operations.

As the user can use the `import` statement in his code, this file contains functions to check the imported modules, as well as a function to check the redefinition of the variable `myList`. (The `selectedList` variable is bound to the `myList` variable in the user's code.) The following imports are forbidden: - `os`: This module provides a way of using operating system-dependent functionality like reading or writing to the file system. - `sys`: This module provides access to some variables used or maintained by the interpreter and to functions that interact with the interpreter. - `subprocess`: This module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. - `shutil`: This module provides a higher-level interface for file operations, such as copying and removing files. - `socket`: This module provides access to the BSD socket interface. - `http`: This module defines classes which implement the client side of the HTTP and HTTPS protocols. - `ftplib`: This module implements the client side of the FTP protocol. - `pickle`: This module implements binary protocols for serializing and deserializing a Python object structure.

Tailwind and DaisyUI

To end this deep dive into the codebase, I will give you some details about the CSS framework used in the project.

To configure the Tailwind frameworks, ensure to properly configure the following files:

1. `tailwind.config.cjs`: This file is used to configure the Tailwind CSS framework. DaisyUI is added as a plugin in this file. DaisyUI is also configured in this file by adding the `daisyui` key. This allows us to define the themes and the dark mode for the application. For more information about the configuration of Tailwind CSS and DaisyUI, please refer to the Tailwind CSS documentation and the DaisyUI documentation.
2. `postcss.config.cjs`: This file is used to configure the PostCSS framework. It is used to process the CSS files and add the necessary prefixes and optimizations.
3. `app.css`: This file is used to import the Tailwind CSS and DaisyUI styles. It is the main CSS file for the application.
4. `+layout.svelte`: In this file, the Tailwind CSS and DaisyUI styles are imported. This file is used to import the CSS files into the Svelte components. It is the main entry point for the CSS files in the application.

After that, you can use the Tailwind CSS and DaisyUI classes in your Svelte components. To do so, please refer to the Tailwind CSS site and the DaisyUI site for more information about the classes and how to use them.

Conclusion

This document provides an overview of the codebase and how to set up the development environment. It also includes a deep dive into the crucial parts of the codebase, including Svelte files, Python code, and the use of Tailwind and DaisyUI. If you have any questions or need help, please feel free to contact me. You can find my contact information in the README file.