999 (pronounced How-full-is-PER21)

This project was realized during the course "System Orient Programming" at the university of Fribourg. Its goal was to estimate the occupancy of different study rooms across the UniFR campus. The project uses the Atom-lite from M5Stack to track the number of surrounding Bluetooth devices to estimate how many people are in a room. This information is then sent to a server, which calculates the occupancy of the room and displays it on a website.

Table of contents

Overview

The (simplified) file structure of the project looks as follows:

```
firmware/
server/
Makefile
README.md
```

The project is separated in two main parts: `firmware` and `server`. Each section resides in its corresponding folder. The `firmware` folder contains all the files needed to program/flash the atom-lite, and the `server` folder contains all the code to build and deploy the central server.

Firmware

Here is a simplified overview of the file structure of the `firmware` folder:

```
firmware/
  src/
  include/
  lib/
  test/
  platformio.ini
  README.md
```

The firmware of the project uses PlatformIO as its build system, and thus shares the typical file structure of a PlatformIO project. The `platformio.ini` file contains all the configurations and dependencies needed for PlatformIO. The `src` folder is the hearth of the program: it is where most of the code is written.

The `include` folder contains the corresponding `.hpp` or `.h` files to the ones found in the `src` folder. The `lib` folder contains any libraries external to the PlatformIO system, it is empty. The `test` folder contains any tests related to the firmware.

The firmware itself is conceptually divided into two parts. The first one contains all the networking function of the chip, and enables it to communicate with the central server. The second part contains all the Bluetooth functionality and counts the number of Bluetooth devices.

TODO: reorganize the code of firmware and explain clearly what each file is supposed to do.

Server

Here is a simplified overview of the file structure of the `server` folder:

```
server/
   app/
       static/
       templates/
           base.html
           index.html
           room.html
       __init__.py
       app.py
       models.py
       routes.py
   instance/
       sqlite.db
   migrations/
   run.py
   requirements.txt
   Dockerfile
   README.md
```

The server is build using Flask as the server engine and uses SQLite for the database and SQL-alchemy for the ORM. The HTML pages are built with Jinja 2.

`run.py` is the entry point to the program, the rest of the code is contained in the `app` folder. The `static` folder contains all the static elements of the webpage, like pictures, JS scrips or stylesheets. `__init.py__` doesn't contain anything, it just declares the `app` folder as a module. `app.py` codes the initialization of the flask server. `models.py` contains all the code defining the tables of the SQLite database the server uses. `routes.py` handles connections made to the server. It defines what the server does on requests (like calculating occupancy when it receives data from the Atom-lite) and renders the HTML templates.

The `templates` folder contains all the Jinja2 template HTML files. `base.html` contains the basic structure of the all the HTML files (all the other HTML templates extend `base.html`). `index.html` contains the contents of the home page. `room.html` contains the contents of the room page.

The `instance` and `migrations` folder are folders required by flask to manage its database. The database itself is located at `instance/sqlite.db`. It is contained in a single file. If you wish interacting with the database directly, I can recommend tools like `sqlite3` or DBeaver Community if you prefer a GUI.

The `requirements.txt` file contains all the dependencies required to launch the Flask server. You can install all the dependencies by running:

```
pip install -r requirements.txt
```

`Dockerfile` contains all the instructions to create a docker container containing the flask app. This simplifies the deployment of the application.

`Makefile`

The entire project is coordinated with the `Makefile` at the root of the project. Multiple targets are used to execute specific tasks without needing a deep understanding of multiple tools.

Here is a list of available targets:

- `build-docs`: generates the documentation using Doxygen.
- `build-firmware`: builds and uploads the firmware to the Atom-lite using PlatformIO.
- `debug-server`: runs the flask server locally in debug mode. You can the view it on http://127.0.0.1:5000/.
- `clean-server-db`: removes all the database and migration-related files.
- `init-server-db`: initializes the database.
- `migrate-server-db`: updates the database. This is necessary if you change anything in `models.py`.
- `build-server-docker`: this calls docker to build the docker image.
- `run-server-docker`: runs the generated docker image locally. You can access it on http://localhost.
- `stop-server-docker`: stop any locally running docker image.
- `save-server-docker`: compresses the docker image to a tar file that you can upload easily to a remote server.
- `upload-remote-image`: uploads the tar file to the remote server (defined in the Makefile).
- `load-remote-docker`: updates the remote docker image with the tar file we just uploaded.
- `run-remote-docker`: runs the docker container containing our app remotely. It is accessible at http://diufvm30.
- `stop-remote-docker`: stops the docker container running remotely (if there is one).
- `populate-remote-db`: this target is used to populate the remote server with junk data according to the `populate-script.sql` file.
- `deploy-server`: this target builds, compresses, uploads, and runs the flask application - it deploys it.

Any target can be called from the root of the project (where the `Makefile` is located) using:

```
make target
```

Tech stack

Technology	Description (ai-generated)	Use in the project
PlatformIO	PlatformIO is an open-source ecosystem for IoT development, offering cross-platform build automation, library management, and serial port monitoring.	Used as the build-system for the firmware part of our project.
Flask	Flask is a lightweight and flexible Python web framework designed for building web applications quickly with minimal code.	Used as the web engine in our server
SQLite	SQLite is a self-contained, serverless, zero-configuration, transactional SQL database engine.	Database we use on our server.
Jinja 2	Jinja 2 is a fast, widely used, and expressive template engine for Python.	The template language in use for creating the web pages.
SQLAlchemy	SQLAlchemy is a powerful and flexible Python SQL toolkit and Object-Relational Mapper that provides a full suite of persistence patterns.	ORM used in the server to interact with the database.
Docker	Docker is a platform for developing, shipping, and running applications in isolated containers.	Used to containerize and deploy our flask application.

Technology	Description (ai-generated)	Use in the project
Doxygen	Doxygen is a documentation generator for C++, C, Java, Python, and other programming languages.	Used to generate all the documentation in the project.

Dependencies

Because both the firmware and the server use python and python packages to build themselves, you can install all the dependencies using the following command:

```
python3 -m venv .env # create a virtual environment to install the packages to
source .env/bin/activate # activate the virtual environment
pip install -r requirements.txt # install all the python dependencies
```

You will also need `make` if you wish to use the `Makefile`:

```
# ubuntu/debian
sudo apt install make
# macos
xcode-select --install # comes pre-packaged with other development tools
# windows: just use wsl
```

You will also need Docker if you need to deploy the server. Please follow the instructions on the docker website (Linux, macOS).

Building

To build the firmware, just run:

```
make build-firmware
```

To launch the server locally, run:

```
make debug-server
```

> Note: you might need to set an environment variable first. Simply do:
>
> ```
> export FLASK_APP="run:create_app"
> ```

To deploy the server, run:

```
make deploy-server
```

If you wish, you can also build the firmware and server directly. To build the firmware, first go to the firmware directory and run:

```
pio run
```

Or run it through the PlatformIO IDE. To run the server directly, you can run:

```
flask --app run:create_app run # and options like --host or --debug
```

Documentation

The documentation is generated with Doxygen. To generate it, run:

```
make build-docs
# or
doxygen doxyfile # if you want to run it directly
```

> Note: you will need Doxygen to generate the documentation. You can install it at https://www.doxygen.nl/download.html or install it from your package manager.

The documentation can then be found on the web page, there is a 'Documentation' link in the footer.