# Instructions and Clarifications

## Standalone App

The standalone application is a Python 3.9 application using no additional libraries. It includes a battery of tests related to each of the scripts. The Pipfile contains the information about the pipenv environment.

I wanted to create this application to be as modular as possible, so that different classes can be refactored, replaced or removed without altering the rest. My intention has been to apply OOP concepts and try to always follow the "principle of least knowledge".

The CLI class may attract attention. This class isolates the logic of the user's interaction with the rest of the objects. The idea was that it could be replaced in the future by any other controller without affecting the code of the other objects. I did this thinking about a future deployment through a server application, like the one I have built for the Full-Stack app.

#### Commands

In the standalone rootfolder where pipfile is run “pipenv install” to install the environment with the dependencies.

In the same folder, with the environment installed and active, these commands can be used:

* Run tests: python -m unittest discover -v
* Run app: python main.py

## Full-Stack APP

I created this application making use of a FastAPI BE and a React FE.

### BACK-END APP

This application is quite minimalistic as I didn’t have the time to implement stricter validation or making it more modular. I just wanted it to be a FastAPI application with Websockets. The decision to make use of Websockets for communicating with the FE is to avoid the need of a DB to store user information. With this approach every user session (which is represented by a different socket connection) stores the user’s session data and then, when session is over the data is cleaned up. This data these data are comprised of a random World, a drone (with its position) and a history. I did not have the time to send the history to the FE and to paint it, though I would have also liked to.

There are different messages that the BE will receive from FE and that it will send, but always handled by a controller that resembles that of the standalone CLI class. My intention was to make the Flight class the same for both apps, representing a more modular architecture. However, I have not had time to create a single Flight object either; you can see that this is different in both applications, I think this is the major flaw of the development. The difference between both is given by the asynchronous needs of the BE application controller.

#### Commands

In the server rootfolder where pipfile is run “pipenv install” to install the environment with the dependencies, the difference between this pipefile and the other is in the FastAPI, Websockets and uvicorn dependencies.

In the same folder, with the environment installed and active, these commands can be used:

* Run app: python main.py

### Front-End APP

Unlike the BE application, here I have not created a minimalist application, but one that is ready to be scaled. The folder structure is one that I have liked for a long time, a feature-based structure. Inside the app there are classic folders such as components, utils, contexts, constants, styles... The difference is in the features folder.

This folder contains each of the functionalities of the app replicating inside each of them the folder structure of "app". In this way a decoupled scheme is achieved, and more modular one too. We also achieve that the folders of the main directory such as "hooks" are not so saturated. A developer would not be afraid to create as many scripts as he needs because of the risk of not finding them later among dozens of others.

On the other hand, I prefer to make use of contexts instead of using other state management libraries, such as Redux. Contexts allow to modularize the global states by features or by parts of the application. However, contexts have the problem of ordering re-rendering to the components that listen to the context each time the object returned by that context is altered.

This major problem of contexts is solved by using the "use-context-selector" library, which allows the use of selectors for components to subscribe only to one needed property inside this context returned object. Thanks to that we avoid those components from re-rendering when a prop changes that they have no connection with.

Additionally I use Vite and TS and SCSS.

I did not have time to develop features like a history display or a landing/takeoff action.

#### COMMANDS

In the drone-fe root folder run “npm install” to install the dependencies inside the package.json.

Then run “npm run dev” to start the development server and the app.