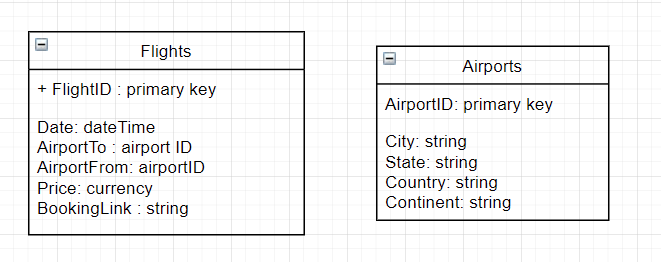
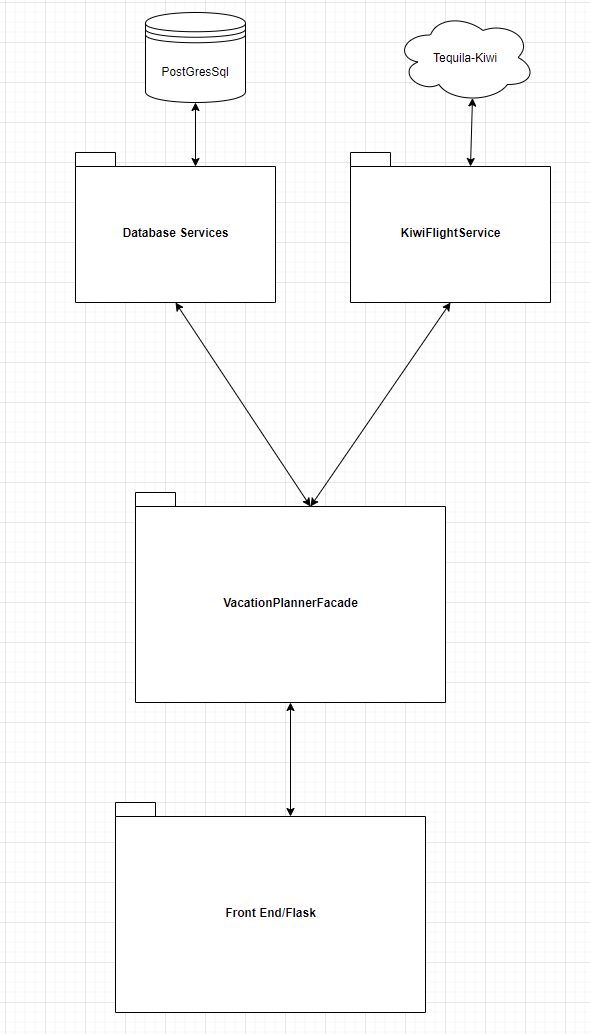
Tomasz Stolarczyk

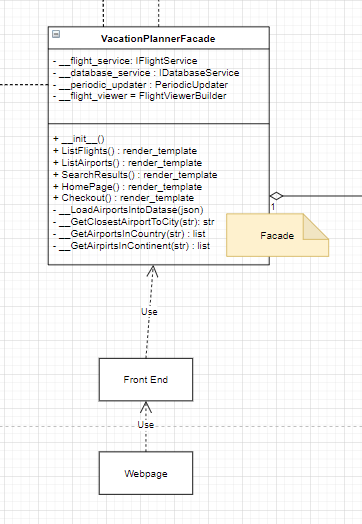
I started off the assignment by working through a good database schema. I wanted a database that will store both Flight info and airport locations. Since the data should have a schema, I needed to also find a relational database. I chose the following database schema. The BookingLink in the Flights is for the api callback to allow the booking of the flight.



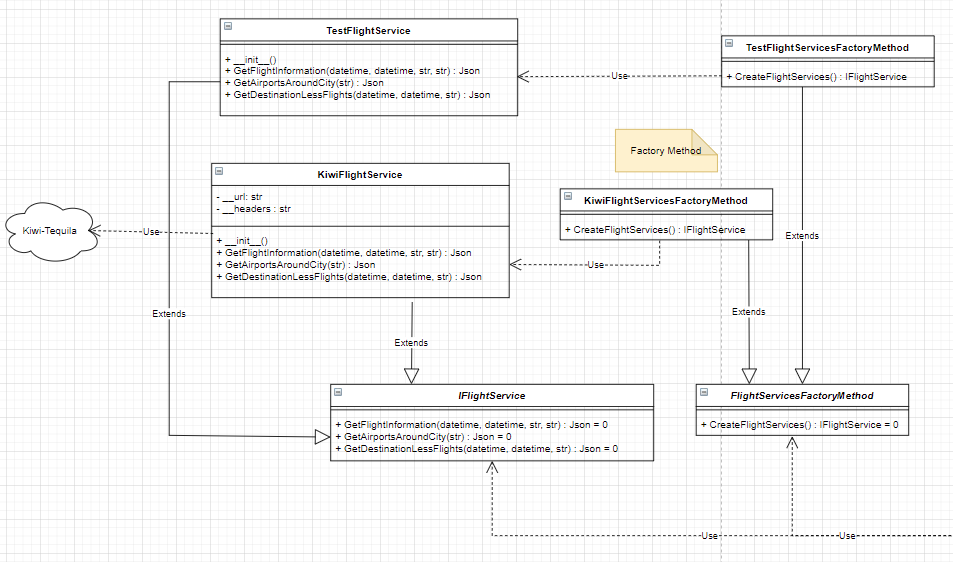
Now that I had a schema, I needed to choose a database system. I chose PostGresql for its modern interface and handy docker containers. I chose to use docker containers to minimize the amount of local setup required.

Using DockerCompose also allows us to instantiate multiple docker containers and manage them from a simple command line. Inside the docker environment, I can have the database in a separate container but still communicate through TCP to get the data.

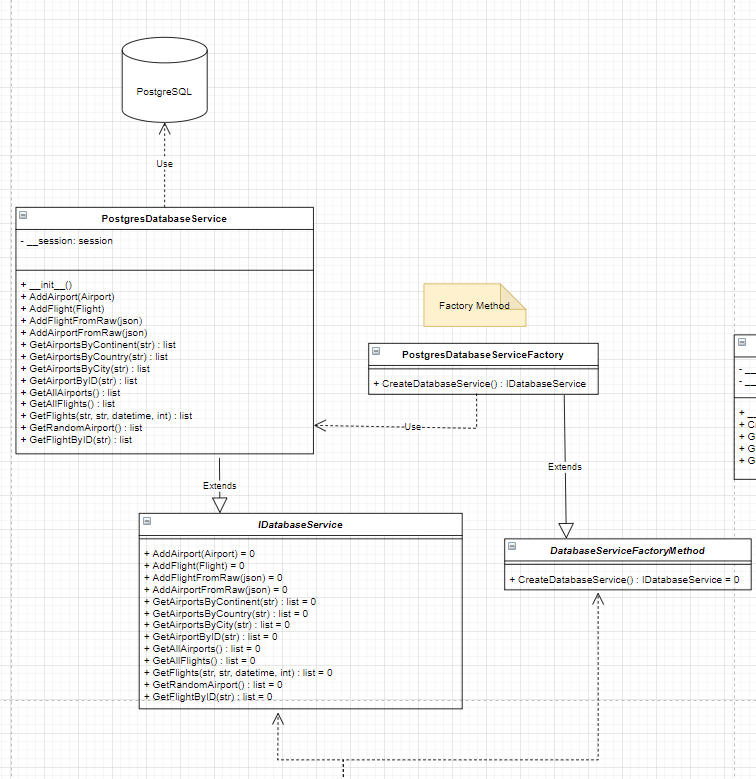
Since there are 2 major systems that I would need to communicate to, namely the web system that holds live flight data and the postgres database which holds potentially stale data, I decided to split up the design in a very broad level. One Service will be responsible for one major system. That way I can decouple them as best I can.

Using Flask and python throughout, Flask hooks up to python through global methods. Since there needs to be a mapping from html layout and the business part of the program, a Façade pattern is perfect for this role. The Façade will also bind the Database Services and the Flight Services together to accomplish the goal of booking/searching flights.

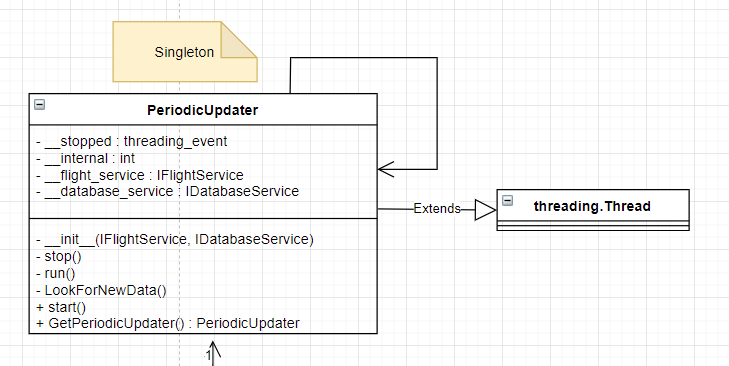
In order to decouple the Flight Services and the Database Services from the rest of the codebase, I decided to utilize the Factory Method pattern. The Factory Method pattern allows me to swap in and out different Flight Service providers and was particularly useful for when I was testing. Since a long portion of the runtime is in querying a flight database and I was worried that I would get kicked out, I saved to file a single query and then created a TestFlightService that was constructed through a TestFlightServiceFactory to quickly pretend to be a real flight service for the façade.



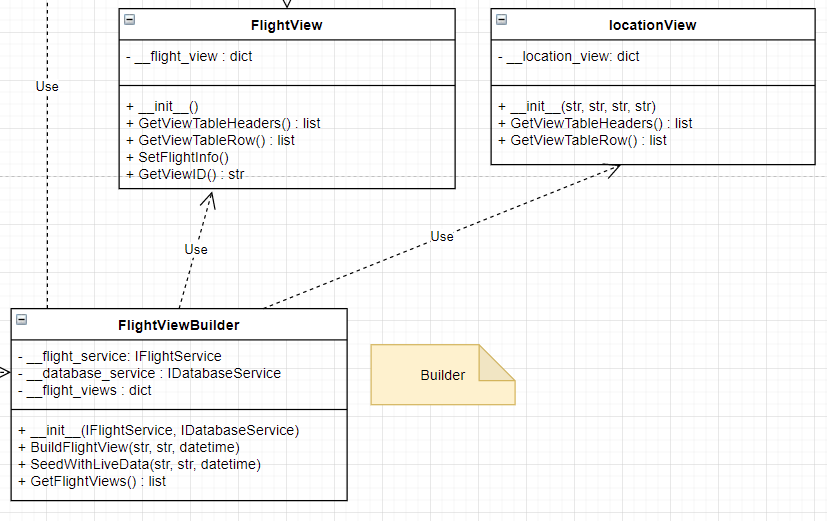
I applied a similar approach for the DatabaseService with the intention of creating more factories as they relate to inmemory databases for quick tests. I can also swap out the underlying database very simply by extending from the IDatabaseService and DatabaseServiceFactoryMethod classes to create a database with an Excel backend, for example.



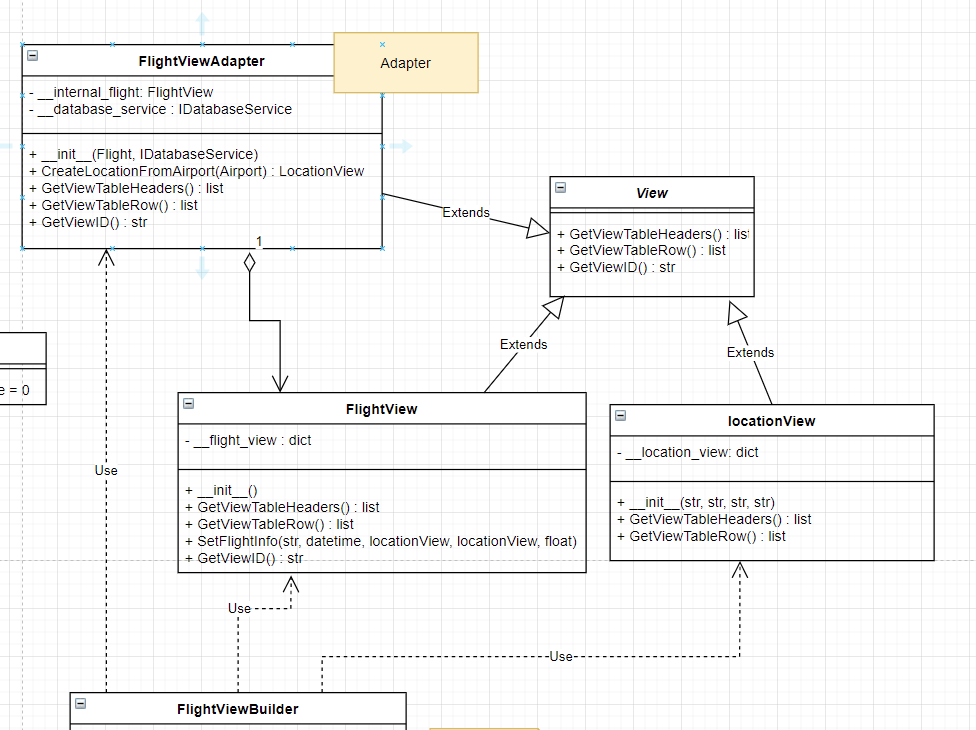
Once I had these 2 services fleshed out and running, I needed data. For this, I would be required to create a periodic thread which every so often will reach out with the Flight Service and grab up to date flight data. This is in hope of getting ahead of any data queries that come in the future, we can save time by querying for these locations in the future. For this, I created the PeriodicUpdater. Since the PeriodicUpdater maps to a single thread and I only want one, it made sense to turn this class into a Singleton. Since I have the ability now to query for any data I wish, I thought it also made sense to expand the existing database and have the vacation planner learn about other airports. For this behavior, the PeriodicUpdater will use the IDatabaseService to randomly choose an already known Airport and use the IFlightService service to query for any outgoing flights from this location. If it meets an airport it’s never seen before, it will store the new airport into it’s database.



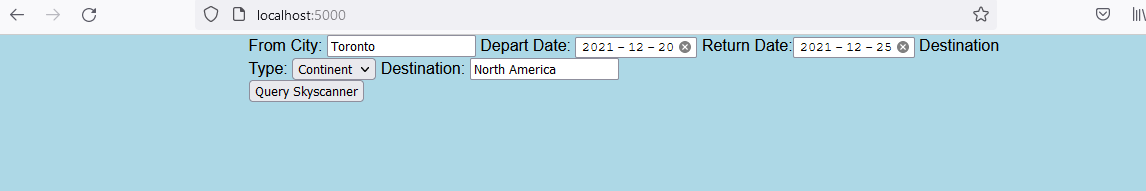
Finally, we must choose a good way to view the results of our search. The Façade will receive the search query, but it would be better if we can delegate the creation of the search result to another class. This class will then be responsible to take some flights from the database and some flights from the live flight service and return a presentable array of flights. Since the creation of this result is non-trivial, I’ll use the Builder pattern to build up the result. The FlightViewBuilder will then build up the FlightView and LocationView classes.

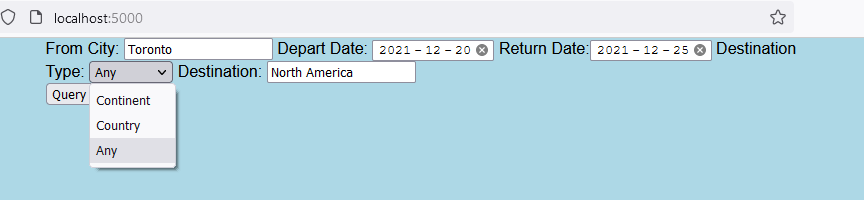


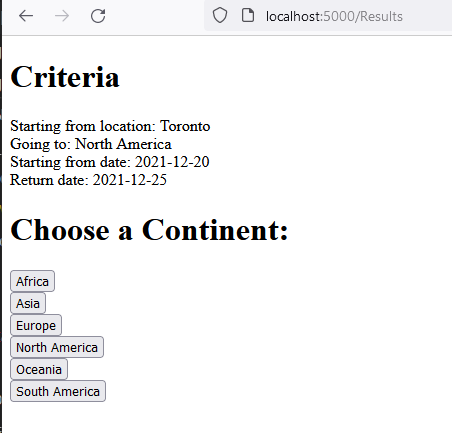
Since we have to deal with a mixture of flight data from the database and the live data, I’ll create an adapter class which adapts the Flight class that we get from the database to the FlightView class. That way, FlightViewBuilder only needs to deal with Views that have a known GetViewTableHeaders() and GetViewTableRow().



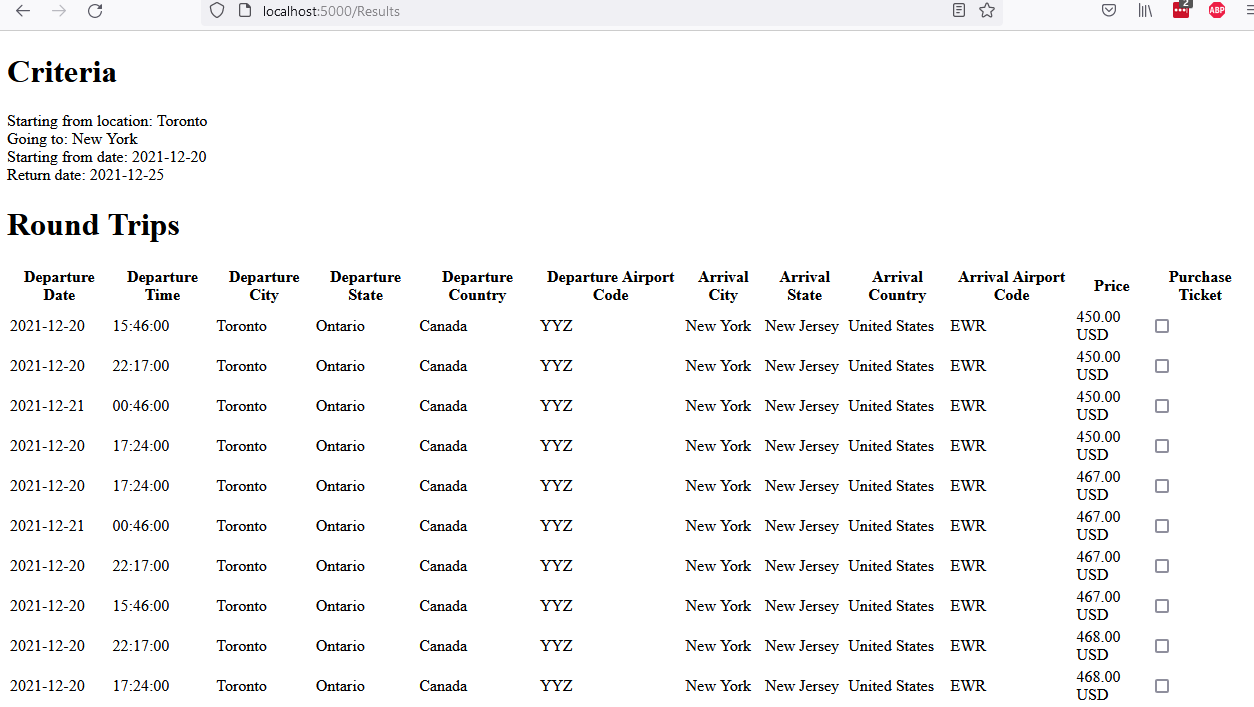
Example run:

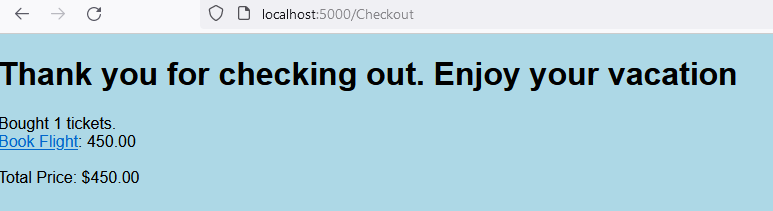


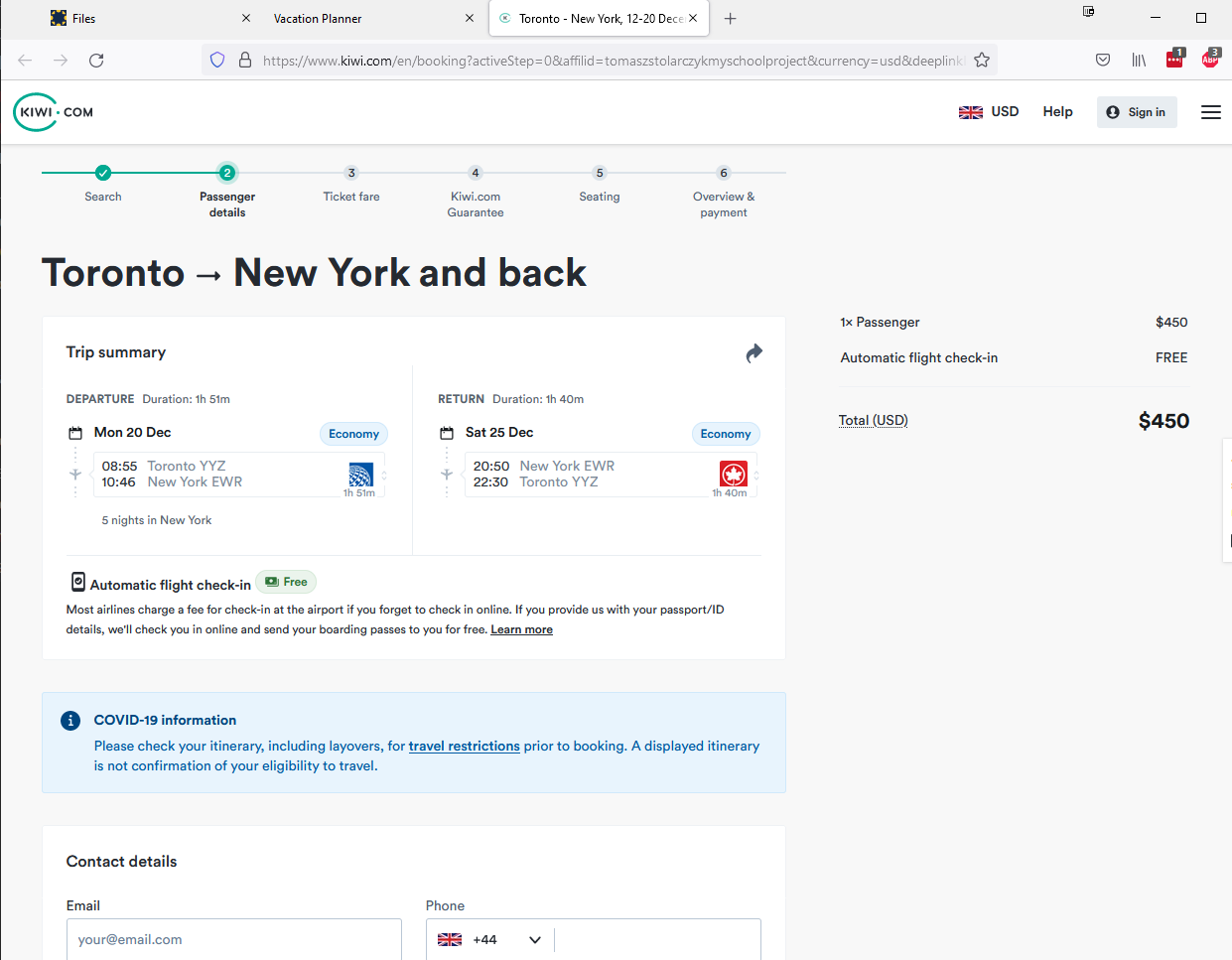
Choosing Any allows you to then choose between the various continents and countries, showing you the available flights.







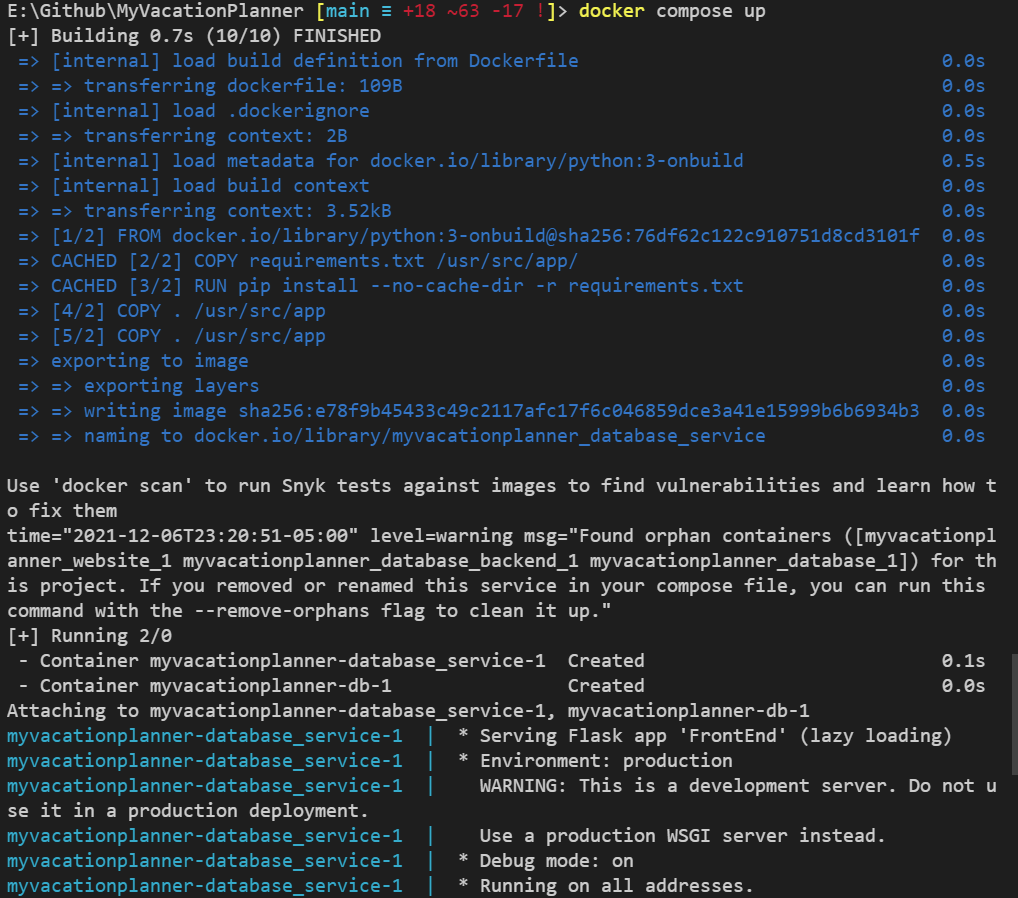




Reference: <https://tequila.kiwi.com>

To Start: Install Docker

Run docker compose up



Navigate to localhost:5000 in a local browser.