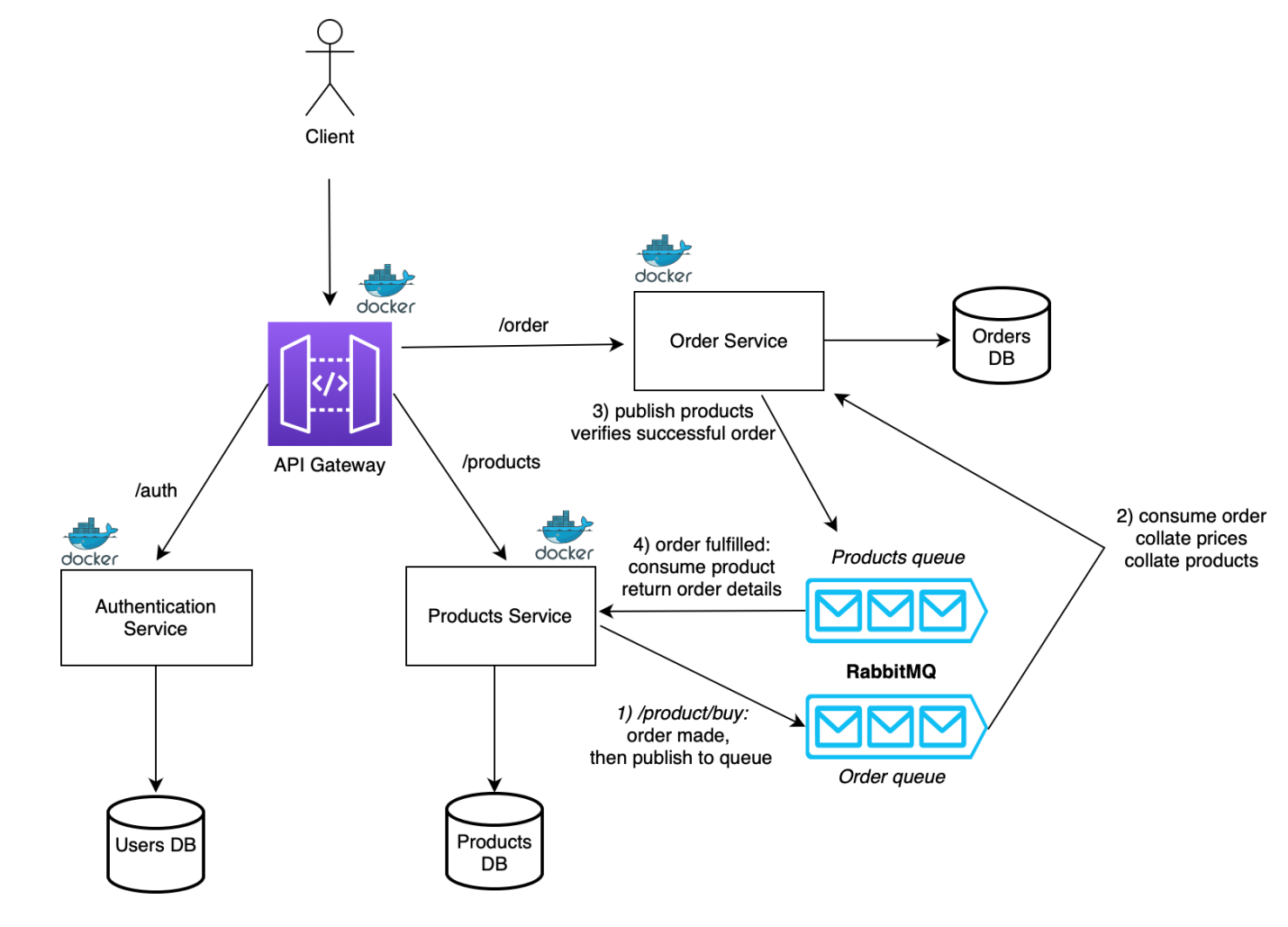
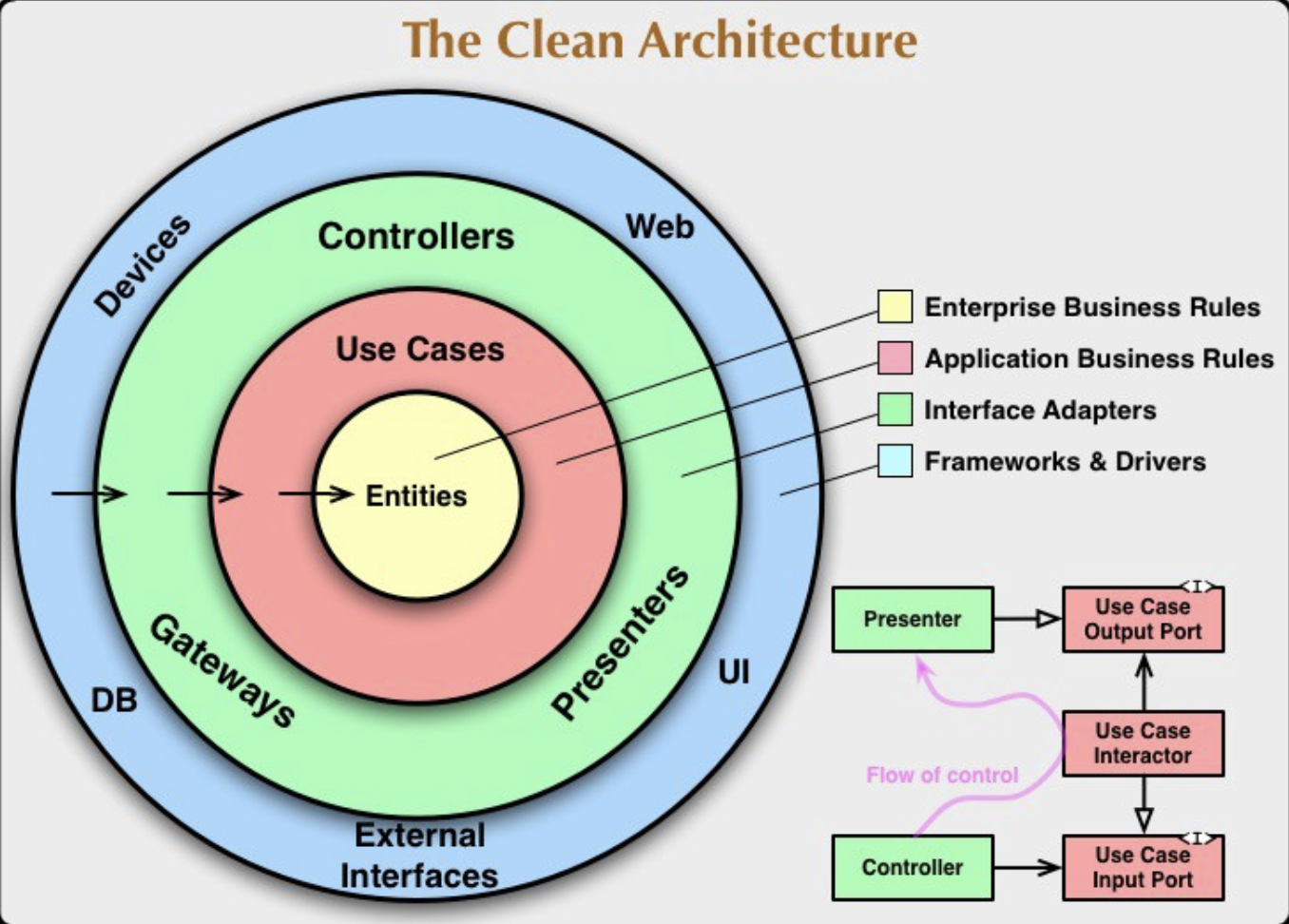
**Software Architecture:**



* The application uses an API gateway to bind all services along a single front, acting as a proxy for the domains in which the auth, order and product microservices are deployed on
* Each microservice, the API gateway and RabbitMQ are deployed as Docker images
* Interactions between product service and order service uses [AMQP](https://www.amqp.org/) protcol, using RabbitMQ which consists of two queues - orders and products. This saves on resources allocated for REST calls to MongoDB.
* product service publishes to the order queue which is then consumed and collated by order service
* order service publishes ordered products to the product queue which is then consumed by product to return order details

**Microservice Architecture:**



* The architecture for a microservice is inspired by Uncle Bob's [Clean Architecture](https://www.freecodecamp.org/news/a-quick-introduction-to-clean-architecture-990c014448d2), which supports strong modularity, loose coupling and dependency injection

# ****Designing the Microservices and API Gateway****

Let’s discuss the microservices that play distinct roles in fulfilling the overall functionality of the system. Each microservice is responsible for a specific domain and communicates with other microservices to accomplish the desired outcomes.

1. **Authentication Microservice**: The authentication microservice handles user authentication and authorization. It ensures that only authorized users can access protected resources within the application. This microservice plays a vital role in securing the entire system and managing user credentials.
2. **Product Microservice**: The product microservice is responsible for managing product-related information, such as product details, inventory, and pricing. It provides APIs for retrieving product information, adding new products, and updating existing products. This microservice interacts with the database to store and retrieve product data efficiently.
3. **Order Microservice**: The order microservice handles the process of creating and managing orders. It facilitates the placement of orders, tracks the status of orders, and calculates the total price. This microservice communicates with the product microservice to retrieve product details and ensure accurate order processing.
4. **API Gateway**: To streamline communication between clients and the microservices, we have implemented an API gateway. The API gateway serves as a single entry point for client applications, abstracting the complexity of the underlying microservices. It handles incoming client requests and routes them to the appropriate microservices based on the requested resources.

Each microservice in the architecture is encapsulated within its own Docker container, providing a lightweight and portable environment for running the service. Containerization offers benefits such as scalability, isolation, and consistent deployments across different environments.

**Prerequisites**

* Have [npm](https://www.npmjs.com/) and [Node.js](https://nodejs.dev/en/) on your machine
* Have [Docker](https://www.docker.com/) installed
* Have [RabbitMQ](https://www.rabbitmq.com/) installed
* Set up your own [MongoDB](https://www.mongodb.com/) collection with appropriate security/credential settings

**Steps to Run**

### On Docker

1. Create a .env file following the format specified in the /auth/env.example, order/env.example and product/env.example directories, following the format specified in each microservice directory
2. Run docker-compose build
3. Run docker-compose up. Now you can test the APIs from localhost:3003

### On localhost

1. Create a .env file following the format specified in the /auth/env.example, order/env.example and product/env.example directories, following the format specified in each microservice directory
2. Run npm install in the /auth, /product, /order and /api-gateway directories
3. Run npm start on all four directories mentioned in the step above. Now you can test the APIs from localhost:3003

## Future work and improvements

* It could be useful to use Kubernetes for container orchestration in order to bundle up this project into one cohesive unit
* While I tried to follow a TDD approach - that is, letting test cases guide development - I eventually gave up on it in the name of speedy development. Ideally, I could have written unit tests first, and slowly increment up to integraton tests and then system tests.
* The internal service of each microservice does not follow pure dependency injection advocated in Clean Architecture. The internal file structures and flow of dependencies are loosely based on Clean Architecture and the code does not fully utilise dependency injection principles. While I did try to minimise interdependencies, I found it a bit overkill to follow Clean Architecture fully for what is essentially a take-home project. But it is worth trying eventually.
* I'd like to write a series of Bash scripts with various curl commands to automate API testing and follow them in sequence of particular use-cases (e.g. user publishes product -> another user logs in -> other user buys product -> ...)
* It could be a good exercise to deploy the databases across different platforms (e.g. Firebase, SQL, etc.) to prevent a single point of failure