Design Decisions and Assumptions

1.**Modularization**:
The code is organized into separate modules such as
`controllers`, `interfaces`, and `routers`,`utilities`, `test`, promoting modularity and maintainability.
This separation allows for clear responsibilities and easier testing.

2. **Controller Functions**:
Each controller function
(`createOrder`, `getOrder`, `getOrderById`, `getPharmacy`)
corresponds to a specific API endpoint.
This separation follows the principle of single responsibility,
making the code easier to understand and manage.

3. **Error Handling**:
Error handling is implemented using try-catch blocks in each controller function.
This ensures that any errors that occur during the execution of the API calls are properly caught and logged,
providing a more robust and reliable application.

4. **Logger Implementation**:
A logger is used to log information and errors throughout the application.
This enhances traceability and aids in debugging. The logger is imported from a separate module, `logging.ts`,
which is implemented using popular logging libraries like Winston.

5. **Pharmacy Order Handler**:
The `PharmacyOrderHandler` class is responsible for creating order payloads based on the type of pharmacy specified. This design decision allows for easy scalability if new pharmacies need to be added in the future.

Each pharmacy has its own order processing logic encapsulated within
private methods (`processHealthMartOrder`, `processCarePlusOrder`,
`processQuickCareOrder`), promoting **encapsulation** and **maintainability**.

6. **Data Validation**:
Basic data validation is performed within the `createOrder` function to ensure that the incoming order payload contains the required fields based on the type of pharmacy specified. If the payload is invalid, an error is thrown indicating an unsupported pharmacy

7. **Axios for HTTP Requests**:
Axios is used for making HTTP requests to external APIs (`pharmacy-mock-service`).
Axios provides a simple and powerful API for making HTTP requests and handling responses asynchronously.

8. **Base URL**:
The base URL for the external pharmacy mock service is defined as a constant (`BASE_URL`).
This allows for easy modification in case the URL changes or needs to be configured differently for different environments (e.g., development, production).

9.**Testing**
Jest is used to create unit test cases and also generate report for the test cases to check coverage for statement, branch and code.

9. **Assumptions**:
- The pharmacy mock service follows RESTful conventions for its API endpoints.
- The pharmacy mock service responds with appropriate status codes and data formats.
- The structure of the order payloads (`OrderPayload`, `HealthMartOrderPayload`, `CarePlusOrderPayload`, `QuickCareOrderPayload`) matches the expected structure defined in the interfaces (`orderInterfaces.ts`).
- The logger implementation (`logging.ts`) is correctly configured to log information and errors to an appropriate destination (e.g., console, file).
- The application is running in an environment where TypeScript is supported and properly configured.