Design Decisions and Assumptions

Here's how the code handles the design requirement of adding new pharmacies without major code changes:

1. **Interface Segregation**:
   The code uses TypeScript interfaces (HealthMartOrderPayload, CarePlusOrderPayload, QuickCareOrderPayload) to define the structure of the order payloads specific to each pharmacy. This allows for clear separation of concerns and easy addition of new pharmacies by simply defining a new interface for the new pharmacy's order payload.
2. **Type Guards**:
   The code uses private methods isHealthMartOrder, isCarePlusOrder, and isQuickCareOrder as type guards. These methods check if the provided OrderPayload adheres to the structure of a specific pharmacy's order payload. If the payload matches the structure of one of the supported pharmacies, TypeScript treats it as that specific type (HealthMartOrderPayload, CarePlusOrderPayload, or QuickCareOrderPayload). This enables the createOrder method to correctly identify the type of order and process it accordingly.
3. **Scalable Switch Statement:**
   The createOrder method uses a switch statement to handle different types of orders. Adding a new pharmacy involves simply adding a new case to this switch statement along with its corresponding type guard. This makes it easy to extend the functionality to support new pharmacies without having to modify existing code significantly.
4. **Flexible Processing Method**:
   The processOrder method extracts relevant data from the order payload based on keys specific to each pharmacy (productKey, quantityKey, customerInfoKey). This method is flexible enough to handle different structures of order payloads for different pharmacies without requiring changes to the method itself.

Lets understand in brief how this is being handled

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.