Summary:

This is a simple implementation of a web application of a digital shopping cart. Similar to a basket on an e-commerce website or at the self-serve checkout in a supermarket.

Design Approach

Architecture:

I went with a layered architecture to implement this backend system. I added these layers to organize the code, separate concerns, and ensure the application is maintainable and scalable. Here's an explanation of the different layers, from the API layer to the database layer:

1. **API Layer** (Presentation Layer): This is the entry point of the backend and is responsible for handling incoming HTTP requests from clients. Its main tasks include:
    - Parsing incoming requests.
    - Orchestrating business logic by calling appropriate services for cart, checkout, item and customer.
    - Formatting and sending responses back to the client.

2. **Service Layer** (Business Logic Layer): The service layer contains the application's core business logic, even though it is pretty simple and lean for this project. It handles data processing and implements the application's rules and workflows. Each domain concern in this layer has an abstraction and its corresponding implementation. This layer is responsible for:
    - Coordinating multiple data sources by fetching from the appropriate data repo.
    - Applying business rules and mapping the data to and from the DTOs.

3. **Data Access Layer**: The data access layer is responsible for interacting with the database. It abstracts away the complexities of database interactions and provides a clean interface for the rest of the application to work with data. Here I used Hibernate, the Java JPA implementation for persistence. Responsibilities include:
    - Retrieving and updating data from/to the database.
    - Executing database queries.
    - Managing database connections and transactions.

4. **Database Layer:** This is where the actual data is stored and managed. My choice is MySQL. The reason for choosing MySQL for this project is the ease of use and flexible administration, all with incredible performance and room for scalability in the future

Notable Addition:

These layers work together in a structured manner to create a separation of concerns and facilitate the maintenance, scalability, and modularity of your application. I tried to design the interactions between these layers carefully, ensuring a clear flow of data and responsibilities while adhering to best practices for software architecture.

Implemented Features

Backend (Functional and Non-functional)

1. I added endpoints to expose creating items, creating customers, adding items to the cart, and removing items from the cart.
2. I added business logic to cover adding and removing items from the cart.
3. I added business logic to check out a user's cart items.
4. I added a table to store offers settings. This allows us to add more offers in the future.
5. I added a **feature toggle** functionality on the 2-for-1 offer. This allows the app manager to control when this offer is available and when it should be disabled. This is controlled with the **active** flag on offers DB table.
6. Test coverage for main business logic around checkout, cart and offer.
7. Containerization of the modules for easy run on the local environment.

Current Drawbacks and Future Improvements

- The current implementation does not have validators to ensure the DTOs have the right data for processing.
- Better exception handling can be implemented and it can be made to be as granular as possible.
- In most real-life scenarios, prices change, therefore it should be removed from the Cart model and be read from the item model at the point of processing checkout.
- The current models connect based on primary-foreign key relationship, we can take advantage of JPA relationship Mappings to further link the models together.
- Heavy integration tests can be added on the API Layer, to test that all layers and modules are well integrated.
- A caching layer can be added to store information that does not often change like offer configuration, product information etc. This will reduce some current round trips to the db each time the information is needed.

Ultimately, by using a layered architecture, these future improvements should be easy to plug in since the system is layered and the concerns adequately separated.