Application Canvas

Name	Order Service	
Description	Provides RESTful API services to Create, View, Cancel and Delete orders	
Service API		
Commands	Synchronous	
	createOrder()	Used to initialize and create orders
	findAllOrders()	Used to view all the present orders from the database
	findOrderById()	Used to view a specific order using orderId
	cancelOrder()	Used to cancel an already placed order
	deleteOrder()	Used to delete an order from the database
Implementation	3-tier Data Model:	
	API layer <-> Service Layer <-> Data Layer	
Observability	all_orders item_details billing_details shipping_details payment	

Database Schema:

The given database in the legacy schema is monolithic and contains all the details in a single table. The first major concern is to normalize the given schema.
The data is split into 5 logical data groups, i.e., 5 tables. This leads to following advantages:

- Logical data division
- Increased readability
- Reduce redundancy
- Greater data organization
- Data consistency

Having said that, the schema is divided into 5 tables:

- billing_details: This table consists of the billing information for items. It has a 1-1 relationship with the *order* table It contains:
    - address_line1 and address_line2
    - city
    - state
    - zipcode
    - item_name
- items: This table consists of the details about the item that is being ordered. The order table has 1-* relationship with *items* table. It contains:
    - item_name
    - item_id
    - item_qty

    - item_price
- orders: This table consists of the crux of this service. It has the details about the orders. It has a 1-* relationship with *item* table, 1-1 with *billing_details* table, *payments* and *shipping_details*. It contains:
    - order_id
    - customer_id
    - customer_name
    - customer_email
    - status
    - created_date
    - modified_date
    - notes
    - order_shipping_charges
    - order_subtotal
    - order_total
    - order_tax
- payments: This table contains the details about the payment methods used by the client in order to place the order. It has a 1 - 1 relationship with the *order* table. The customers can split the billing amount into different payment methods, i.e., CreditCard, DebitCard, PayPal, etc. by adding this feature in the service layer. It contains:
    - payment_method
    - payment_date
    - payment_confirmation_number
    - payment_card_number
    - payment_cvv
- shipping_details: This table contains the shipping details for the order. It has 1-1 relationship *order* table. It contains:
    - shipping_method
    - address_line_1
    - address_line_2
    - city
    - state
    - zipcode

REST API Design:

- After designing the database, next step is to design the API and its end points. The API design is divided into 3 sections:
    - API layer
    - Service Layer
    - Data layer
- The API layer is responsible for all the CRUD operations, i.e., defining the end points. The operations are defined in the OrderController class. This class will further use the OrderService class to that is in turn connected to the repository that extends the JpaRepository used to connect to the Postgres Database in the backend.
- There are several annotations that are used in Spring to perform several operations in the background without actually writing everything from scratch.
- The prominent annotations in this application are:
    - @Service: Class level annotation responsible for letting marking a class that performs some service
    - @Entity: Tt specifies that the class is an entity and is mapped to a database table

    - @RestController: It is used to create RESTful web services
    - @RequestMapping: It specifies the URI for the end point
    - @GetMapping: It is used to map GET request to specific handler methods
    - @PostMapping: It is used to map POST request to specific handler methods
    - @Autowired: It is used for object dependency injection
    - @Table: It specifies the name of the database table that is to be used for mapping
    - @OneToOne(and other relationships): It is used to specify relationships between fields
- The root for the RestController is “/api”. The various endpoints exposed in this application are:
    - GET: findAllOrders ("/orders"):
        - returns a list of orders in the Order table
    - GET: findByOrderId ("/orders/{orderId"}):
        - This route takes in an *orderID* and returns the Order associated with that OrderId
    - POST: createOrder ("/orders/create"):
        - This takes in the RequestBody from the POST request and saves that order
        - Save method in the OrderService class, will set the createdDate to UTC
        - It will also save the Status on the order as RECEIVED
        - Example:

```json
{
  "customerId": "6",
  "customerName": "Dom",
  "customerEmail": "dom@gmail.com",
  "orderShippingCharges": 100.0,
  "notes":"Delivery on Wednesday",
  "items": [
    {
      "itemName": "Alexa",
      "itemQty": 5,
      "itemPrice": 50,
      "itemId": 17
    },{
      "itemName": "Nest",
      "itemQty": 20,
      "itemPrice": 45,
      "itemId": 18
    }
  ],
  "shippingDetails": {
    "addressLine1": "Unit 123",
    "addressLine2": "First corner",
    "shippingMethod": "FedEx",
    "city": "Mumbai",
    "state": "Maharashtra",
    "zipcode": "9757574"
  },
  "payments": {
    "paymentMethod": "PayPal",
    "paymentConfirmationNumber": "121",
    "paymentCardNumber": "123123123123",
    "paymentCardCVV": "123"
  },
  "billingDetails": {
    "addressLine1": "Unit 234",
    "addressLine2": "Second corner",
    "city": "Mumbai",
    "state": "Maharashtra",
    "zipcode": "9757574"
  }


}
```

- Output:

```
{
  "orderId": 10,
  "customerId": "6",
  "customerName": "Dom",
  "customerEmail": "dom@gmail.com",
  "status": "RECEIVED",
  "createdDate": "2021-06-09T06:05:12.724+00:00",
  "modifiedDate": null,
  "notes": "Delivery on Wednesday",
  "orderShippingCharges": 100.0,
  "orderSubtotal": 1150.0,
  "orderTotal": 1230.5,
  "orderTax": 80.50000000000001,
  "payment": null,
  "billingDetails": null,
  "shippingDetails": {
    "shippingDetailsId": 6,
    "shippingMethod": "FedEx",
    "addressLine1": "Unit 123",
    "addressLine2": "First corner",
    "city": "Mumbai",
    "state": "Maharashtra",
    "zipcode": "9757574"
  },
  "items": [
    {
      "itemId": 17,
      "itemName": "Nest",
      "itemQty": 20,
      "itemPrice": 45
    },
    {
      "itemId": 18,
      "itemName": "Alexa",
      "itemQty": 5,
      "itemPrice": 50
    }
  ]
}
```

  - GET: cancelOrder ("/orders/cancel/{orderId}")
    - This method will update the Status of the existing to CANCELED
  - GET: deleteOrder ("/orders/delete/{orderId}")
    - This endpoint receives the *orderId*, if it exists, deletes that order record

Logging:

- Lombok Slf4j package dependency is used to log information in the console
- It is easier and helps in debugging and troubleshooting the code
- Logging also helps in keeping check on the activities performed by the application for future reference
- It helps in identifying records from the database using orderId, customerId, itemId, customerName, etc.

Test:

- Initial phase of testing done using Postman. Setting inputs and checking for desired outputs.
- Added 4 test classes to test the application:
  - Check whether the connection to the server is well established
  - Check whether the endpoint is reachable
  - Check whether the context is creating the controller, i.e. whether the @Autowired annotation is functional
  - Check whether the given input does return an expected output as JSON
- There can be multiple tests that can be added to make the application more tolerant