**Price Service**

The **Price Service** is responsible for managing dynamic pricing for products in an e-commerce platform. It ensures price configurations are unique for each combination of productId, sku, and contextual attributes (Customer Group, Country, and Currency). This service should allow creating, updating, deleting, and retrieving prices dynamically based on these attributes. Below are detailed instructions for students to implement the service.

**Step 1: Define the Price Entity**

1. **Purpose**:
   * The Price entity represents a single price configuration for a product.
   * Each price configuration is tied to a specific productId and sku, and optionally to contextual attributes like customerGroup and country.
2. **Fields to Include**:
   * id (Primary key)
   * productId (Reference to the product this price belongs to)
   * sku (Stock Keeping Unit, unique identifier for variants of a product)
   * customerGroup (Optional, identifies specific customer groups)
   * country (Optional, defines geographic-specific pricing)
   * currency (Required, ensures pricing is currency-specific)
   * priceAmount (The actual price of the product)
3. **Unique Constraint**:
   * Ensure the combination of productId, sku, customerGroup, country, and currency is unique.
   * This prevents duplicate configurations.
4. **Implementation Notes**:
   * Use JPA annotations to map the entity to the database table.
   * Add a table-level unique constraint for the combination of fields mentioned above.

**Step 2: Create the Price Repository**

1. **Purpose**:
   * The repository handles database interactions for the Price entity.
   * It should support operations like adding, updating, deleting, and fetching prices.
2. **Required Methods**:
   * **Fetch Prices for a Product**:
     + Retrieve all price configurations for a specific productId.
   * **Fetch Price Dynamically**:
     + Query price dynamically based on productId, sku, and contextual attributes.
   * **Check for Duplicate Configuration**:
     + Add a method to check if a price configuration already exists with the same productId, sku, customerGroup, country, and currency.
3. **Logic for Checking Duplicates**:
   * Use a custom query in the repository to verify if a configuration exists before saving:
     + Match all fields (productId, sku, customerGroup, country, currency) to check for duplicates.

**Step 3: Design the PriceRequest and PriceResponse DTOs**

1. **Purpose**:
   * **PriceRequest**: Used to capture inputs for creating or updating price configurations.
   * **PriceResponse**: Used to return price data in API responses.
2. **Fields in PriceRequest**:
   * productId, sku, customerGroup, country, currency, priceAmount.
3. **Fields in PriceResponse**:
   * Same as PriceRequest, including any additional metadata if needed (e.g., id).
4. **Validation**:
   * Ensure productId, sku, currency, and priceAmount are mandatory fields.
   * Optional fields (customerGroup, country) should support null values.

**Step 4: Implement the Pricing Service Logic**

1. **Purpose**:
   * Encapsulate business logic for adding, updating, deleting, and fetching prices.
   * Ensure no duplicate price configurations are allowed.
2. **Core Methods**:
   * **Add Price**:
     + Check if a price configuration with the same productId, sku, customerGroup, country, and currency already exists.
     + If a duplicate exists, throw an exception.
     + Save the new price in the database.
   * **Update Price**:
     + Fetch the existing price by its ID.
     + Ensure the updated configuration does not create a duplicate.
     + Update the fields and save changes.
   * **Delete Price**:
     + Remove the price by its ID.
   * **Fetch Prices by Product**:
     + Retrieve all prices for a specific productId and optionally filter by sku.
   * **Fetch Dynamic Price**:
     + Query prices based on productId, sku, and contextual attributes.
     + Use a priority order for attributes:
       1. Customer Group + Country + Currency
       2. Customer Group + Currency
       3. Country + Currency
       4. Currency
     + Ensure a price is returned even if some attributes are missing.

**Step 5: Design the Price Controller**

1. **Purpose**:
   * Expose REST API endpoints for managing price configurations.
2. **Endpoints**:
   * **POST /prices**:
     + Accept a PriceRequest object to add a new price.
   * **PUT /prices/{id}**:
     + Accept a PriceRequest object to update an existing price.
   * **DELETE /prices/{id}**:
     + Remove a price configuration by ID.
   * **GET /prices/{productId}**:
     + Retrieve all prices for a product by productId.
   * **GET /prices/dynamic**:
     + Fetch a dynamic price for a product based on productId, sku, and contextual attributes.
3. **Validation in Controller**:
   * Validate input fields using annotations.
   * Return appropriate HTTP status codes (e.g., 400 Bad Request for validation errors).

**Step 6: Implement Validation and Error Handling**

1. **Validation**:
   * Ensure productId, sku, currency, and priceAmount are always provided in requests.
   * Validate unique constraints during add and update operations.
2. **Error Handling**:
   * Handle exceptions like duplicate configuration, price not found, or invalid inputs.
   * Return meaningful error messages to the API client.

**Step 7: Testing**

1. **Unit Testing**:
   * Test individual methods in the service class to ensure correctness.
   * Write test cases for duplicate configuration checks, adding new prices, updating prices, and fetching prices dynamically.
2. **Integration Testing**:
   * Test API endpoints using tools like Postman or REST Assured.
   * Verify that the service prevents duplicate price configurations and handles dynamic price lookups correctly.
3. **Edge Case Testing**:
   * Test with null or missing values for optional fields.
   * Verify fallback mechanisms for dynamic price retrieval when certain attributes are not provided.

**Fallback Logic for Dynamic Pricing**

1. The service should prioritize attributes in the following order:
   * Customer Group + Country + Currency
   * Customer Group + Currency
   * Country + Currency
   * Currency
2. Use a query to return prices based on this priority, ensuring a fallback price is always returned if possible.