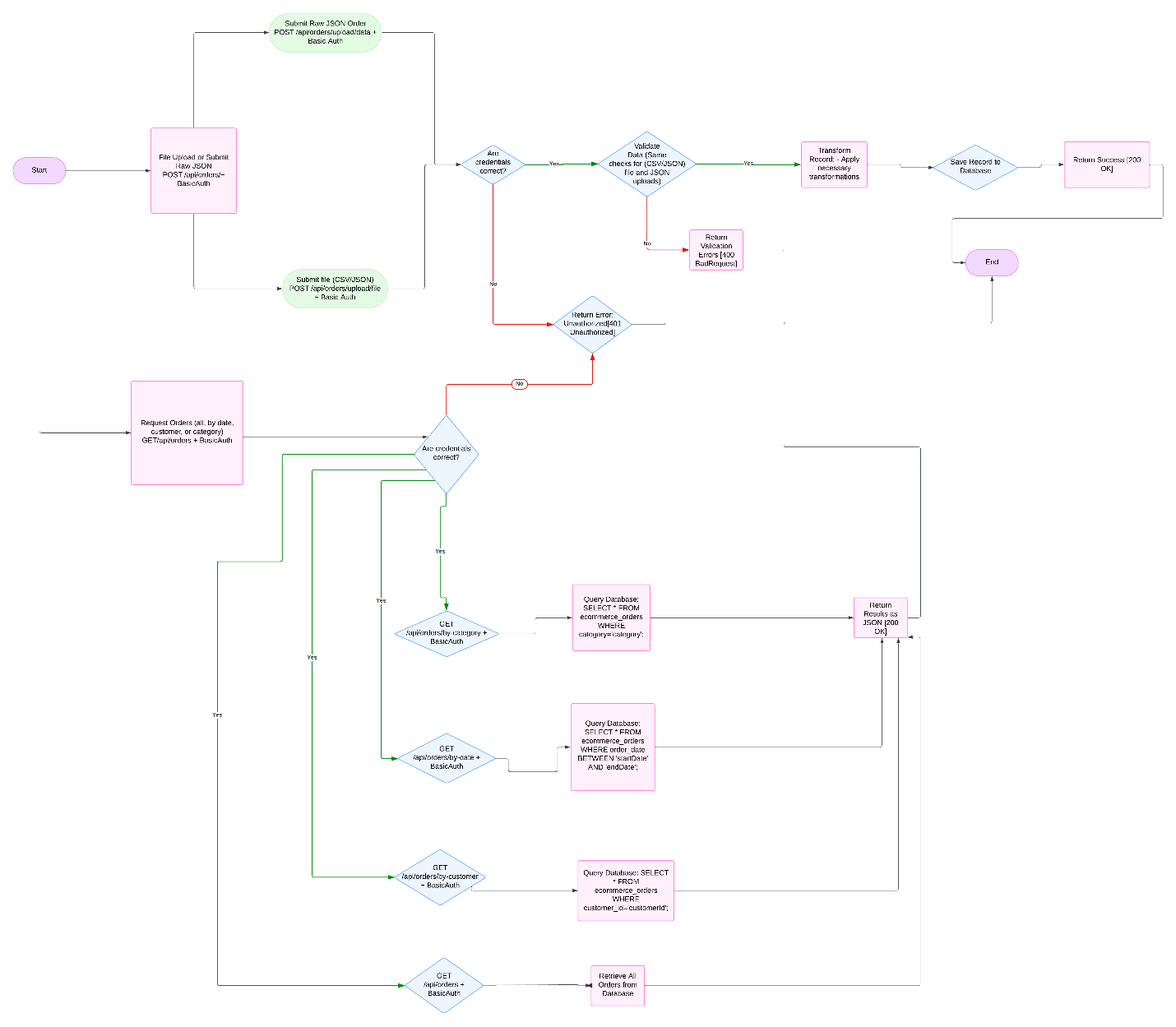
# Cloud-Based Data Processing System

## **System Data Flow Diagram:**



Below is the data flow diagram illustrating how data moves through the system, from ingestion to storage and retrieval:

* **File Upload**: Users upload files (CSV/JSON) containing order data through REST APIs.
* **Validation**: Ensures data complies with constraints such as non-null fields and positive values.
* **Service**: Parses the data, validates it, applies transformations, and prepares it for storage.
* **Transformation**: Calculates total price, applies discounts, and reformats dates.
* **Database (MySQL)**: Stores the processed and validated data.
* **Retrieval**: Data is fetched through REST APIs for different queries (e.g., by customer, category, or date).

## **2. Explanation**

### **a. Security Configuration**

The application is secured using Spring Security. Below are the key details of the configuration:

#### **SecurityConfig Class**

1. **UserDetailsService**:
   * Defines in-memory authentication with a single user.
   * **Username**: user
   * **Password**: password (NoOpPasswordEncoder used for simplicity; not recommended for production).
   * Role: USER.
   * @Bean  
     public UserDetailsService userDetailsService() {
   * UserDetails user = User.withUsername("user")
   * .password("{noop}password")
   * .roles("USER")
   * .build();
   * return new InMemoryUserDetailsManager(user);
   * }

**2. SecurityFilterChain**:

* + Configures HTTP Basic Authentication.
  + **Authentication Rules**:

/api/\*\*: Requires authentication.

Other endpoints: Open to all (adjustable based on application needs).

* + CSRF is disabled for simplicity but can be enabled as needed for enhanced security.

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

return http

.csrf(csrf -> csrf.disable())

.authorizeHttpRequests(auth -> auth

.requestMatchers("/api/\*\*").authenticated()

.anyRequest().permitAll()

)

.httpBasic(Customizer.withDefaults())

.build();

}

### **b. REST API for Data Ingestion and Retrieval**

#### **Ingestion APIs**

1. **POST /api/orders/upload/file**
   * **Description**: Accepts CSV/JSON files for bulk order ingestion.
   * **Request**: Multipart file upload.

Header: Content-Type: multipart/form-data

Body: File upload containing orders in either CSV or JSON format.

* + **Response**:

200 OK: File processed successfully.

400 Bad Request: Validation errors occurred during file processing.

* + **Details of File Processing**:

CSV Files: Parsed using OpenCSV. Each row is mapped to EcommerceOrder fields.

JSON Files: Parsed using Jackson to deserialize JSON into EcommerceOrder objects.

Supported File Extensions: .csv and .json.

1. **POST /api/orders/upload/data**

* **Description**: Accepts a single order as JSON for manual ingestion.
* **Request**: JSON body containing order details.

{

{

"orderId": "123",

"orderDate": "2023-12-01",

"customerId": 1,

"customerName": "John Doe",

"product": "Laptop",

"category": "Electronics",

"quantity": 2,

"unitPrice": 1200.50

}

* **Response**:  
  201 Created: Order successfully saved.

400 Bad Request: Validation errors occurred.

#### **Retrieval APIs**

1. **GET /api/orders**

* **Description**: Fetches all orders.
* **Response**:

200 OK: Returns a list of all orders.

1. **GET /api/orders/by-date**

* **Description**: Retrieves orders based on date filters.
* **Query Parameters**:

filter: Specifies the type of date filter (last\_n\_days, last\_n\_weeks, last\_n\_months, date\_range).

value: Number of days/weeks/months for the filter (required for filters like last\_n\_days).

startDate and endDate: Start and end dates for date\_range filter (both required).

Example:

/api/orders/by-date?filter=last\_n\_days&value=7

/api/orders/by-date?filter=date\_range&startDate=2023-12-01&endDate=2023-12-31

* + **Response**:

200 OK: Returns orders matching the filter.

400 Bad Request: Invalid or missing parameters.

1. **GET /api/orders/by-customer**
   * **Description**: Fetches orders by customer ID.
   * **Query Parameter**: customerId (e.g., /api/orders/by-customer?customerId=1).
   * **Response**:
   * 200 OK: Returns orders for the specified customer.
2. **GET /api/orders/by-category**
   * **Description**: Fetches orders by product category.
   * **Query Parameter**: category (e.g., /api/orders/by-category?category=Electronics).
   * **Response**:

200 OK: Returns orders for the specified category.

### **c. Script/Service for Data Transformation**

#### **DataIngestionService**

* **Responsibilities**:
  + Parses uploaded files (CSV or JSON) using OpenCSV or Jackson.
  + Validates data fields based on constraints (e.g., positive quantities, non-null fields).
  + Applies transformations:

Reformats dates to MM/dd/yyyy.

Calculates totalPrice as quantity × unitPrice.

* + Applies discounts:

15% for orders > $500.

10% for orders > $100.

No discount otherwise.

* **Error Handling**:
* Throws RuntimeException for unsupported file formats or parsing errors.
* Captures validation errors for individual records and provides detailed error responses.

#### **EcommerceOrderService**

* **Responsibilities**:
  + Handles operations for raw JSON order data.
  + Saves individual orders to the database after transformations:

Reformats orderDate to MM/dd/yyyy. If missing, assigns the current date.

Computes totalPrice using quantity × unitPrice.

Applies discount logic:

15% for orders > $500.

10% for orders > $100.

Calculates totalPriceAfterDiscount.

* + **Validation**:

Ensures non-null fields (e.g., quantity, unitPrice).

Prevents negative values for prices and quantities.

* + **Helper Methods**:

reformatDate: Converts date from yyyy-MM-dd to MM/dd/yyyy.

* **Example Workflow**:
  1. Raw JSON order data is passed to saveOrder.
  2. The method validates and transforms the data.
  3. The order is saved to the database using the repository layer.

### **d. Database (MySQL)**

* **Table Schema**:

| Column Name   | Data Type  | Constraints       | Description                                                            |

| id                     | BIGINT          | AUTO\_INCREMENT PRIMARY KEY    | Unique identifier for each record, automatically incremented.            |

| order\_id               | VARCHAR(255)    | NOT NULL                      | Unique identifier for each order.                                        |

| order\_date             | DATE            |                               | The date when the order was placed.                                      |

| customer\_id            | BIGINT          | NOT NULL                      | Unique identifier for the customer who placed the order.                 |

| customer\_name          | VARCHAR(255)    | NOT NULL                      | Name of the customer.                                                    |

| product                | VARCHAR(255)    | NOT NULL                      | Name of the product purchased.                                           |

| category               | VARCHAR(255)    | NOT NULL                      | Category of the product (e.g., Electronics, Accessories).                |

| quantity               | INT             | NOT NULL                      | Number of units purchased.                                               |

| unit\_price             | DECIMAL(10, 2)  | NOT NULL                      | Price per unit of the product.                                           |

| total\_price            | DECIMAL(10, 2)  |                               | Total price for the order before applying any discounts.                 |

| discount               | DECIMAL(10, 2)  |                               | Discount applied to the total price of the order.                        |

| reformat\_order\_date    | VARCHAR(255)    |                               | Reformatted string version of the `order\_date` for display purposes.     |

| total\_price\_after\_discount | DECIMAL(10, 2) |                            | Final price after applying the discount.                                 |

* **Reason for Choosing MySQL**:
  + **Structured Dataset**: MySQL is highly effective for datasets like e-commerce orders, which have a structured format with defined fields (e.g., orderId, customerId, orderDate).
  + **Indexing and Query Optimization**: Advanced indexing capabilities make it easier to query large datasets by frequently accessed columns such as orderDate, customerId, and category.
  + **Scalability for Reads and Writes**: MySQL supports horizontal scaling for read-heavy workloads via read replicas and can handle significant write operations efficiently.
  + **Transaction Management**: ACID-compliant transactions ensure data integrity, which is critical for financial and order-related operations.
  + **Integration with Hibernate and Spring Boot**: MySQL integrates seamlessly with ORM tools like Hibernate and frameworks like Spring Boot, reducing development time and effort.
  + **Partitioning**: MySQL’s partitioning allows splitting large tables (e.g., orders) into smaller, more manageable chunks for improved query performance.
* **Why MySQL for E-Commerce Orders**:
  + The dataset consists of structured fields like orderId, quantity, and unitPrice, making a relational database an ideal choice.
  + Complex queries such as filtering by date ranges or customer IDs are common, and MySQL’s query optimization ensures efficient execution.
  + The need to maintain relationships (e.g., between orders and customers) is effectively handled using foreign key constraints.
  + High data volume and frequent read/write operations can be managed efficiently using MySQL’s scalability features.

### **e. Scaling for 10x Data**

To handle 10 times more data, particularly during retrieval of large datasets, the following scalability strategies can be implemented:

* **Pagination**:

Divide results into smaller pages to retrieve a limited number of records per request. This reduces memory usage and improves performance for large datasets.

* **Filtering**:

Allow users to filter data by parameters such as date, customer ID, or category. This reduces the dataset size returned in queries.

* **Lazy Loading**:

Fetch related data (e.g., associated customer details) only when needed instead of eagerly loading all data. This minimizes unnecessary data retrieval.

* **Caching**:

Cache frequently accessed data to reduce database load and improve response times. Cached results can serve repeated requests faster.

* **Indexing**:

Add database indexes on frequently queried fields like orderDate, customerId, and category. Indexing accelerates search and filter queries.

* **Partitioning**:

Split the database table into partitions to manage large datasets efficiently. For example, partition data by date ranges or customer IDs.

* **Asynchronous Data Streaming**:

Stream data incrementally to the client rather than fetching all records at once. This is useful for handling very large datasets in real-time.

* **Data Archiving**:

Archive older data to reduce the size of the primary dataset. Archived data can be stored separately and accessed only when needed.

* **Load Balancing**:

Distribute incoming requests across multiple application instances. This helps in handling higher traffic efficiently.

By combining these strategies, the system can handle large data volumes efficiently while ensuring optimal performance and a seamless user experience.

## **3. Optional Stretch Goal**

### **a. Data Visualization**

* **Dashboard**:
  + Create a simple frontend dashboard using Chart.js or D3.js.
  + Visualize metrics like total orders, revenue, and category-wise distribution.
* **Downloadable Reports**:
  + Provide an API endpoint to generate and download reports as CSV or Excel files.

## **4. Testing**

### **API Testing Strategy**

* **Unit Testing**:
  + Use JUnit to test service methods like parseCsv, parseJson, and validateAndSave.
* **Integration Testing**:
  + Use MockMvc to test end-to-end API functionality with actual data inputs.
* **File Upload Tests**:
  + Test with valid and invalid CSV/JSON files.
  + Verify behavior with large files, missing fields, and incorrect formats.
* **Validation Tests**:
  + Ensure constraints (e.g., positive quantities, non-null fields) are enforced.
* **Performance Testing**:
  + Use Apache JMeter to simulate high loads and measure response times.
* **Error Handling Tests**:
  + Test edge cases like unsupported file formats, empty files, and duplicate records.

### **Testing Examples for Endpoints**

#### **1. POST /api/orders/upload/file**

* **Input (CSV)**:

Order ID,Order Date,Customer ID,Customer Name,Product,Category,Quantity,Unit Price

101,2025-01-15,1,John Doe,Laptop,Electronics,2,1200.50

102,2025-01-16,2,Jane Doe,Smartphone,Electronics,3,800.00

* **Input (JSON)**:

[

{

"orderId": "103",

"orderDate": "2025-01-17",

"customerId": 3,

"customerName": "Alice",

"product": "Headphones",

"category": "Accessories",

"quantity": 5,

"unitPrice": 100.00

}

]

* **Expected Response**:

200 OK: Orders processed successfully.

#### **2. POST /api/orders/upload/data**

* **Input(JSON)**:

{

"orderId": "104",

"orderDate": "2025-01-18",

"customerId": 4,

"customerName": "Bob",

"product": "Tablet",

"category": "Electronics",

"quantity": 1,

"unitPrice": 300.00

}

* **Expected Response**:

201 Created: Order saved with calculated totals and discounts.

#### **3. GET /api/orders**

* **Expected Response**:

[

{

"orderId": "101",

"orderDate": "01/15/2025",

"customerId": 1,

"customerName": "John Doe",

"product": "Laptop",

"category": "Electronics",

"quantity": 2,

"unitPrice": 1200.50,

"totalPrice": 2401.00,

"discount": 360.15,

"totalPriceAfterDiscount": 2040.85

},

{

"orderId": "102",

"orderDate": "01/16/2025",

"customerId": 2,

"customerName": "Jane Doe",

"product": "Smartphone",

"category": "Electronics",

"quantity": 3,

"unitPrice": 800.00,

"totalPrice": 2400.00,

"discount": 360.00,

"totalPriceAfterDiscount": 2040.00

}

]

**4. GET /api/orders/by-date**

* **Query Parameters**:

Example: /api/orders/by-date?filter=last\_n\_days&value=7

* **Expected Response**:

[

{

"orderId": "101",

"orderDate": "01/15/2025",

"customerId": 1,

"customerName": "John Doe",

"product": "Laptop",

"category": "Electronics",

"quantity": 2,

"unitPrice": 1200.50

}

]

**5. GET /api/orders/by-customer**

* **Query Parameters**:

Example: /api/orders/by-customer?customerId=1

* **Expected Response**:

[

{

"orderId": "101",

"orderDate": "01/15/2025",

"customerId": 1,

"customerName": "John Doe",

"product": "Laptop",

"category": "Electronics",

"quantity": 2,

"unitPrice": 1200.50

}

]

#### **6. GET /api/orders/by-category**

* **Query Parameters**:

Example: /api/orders/by-category?category=Electronics

* **Expected Response**:

[  
{

"orderId": "101",

"orderDate": "01/15/2025",

"customerId": 1,

"customerName": "John Doe",

"product": "Laptop",

"category": "Electronics",

"quantity": 2,

"unitPrice": 1200.50

},

{

"orderId": "102",

"orderDate": "01/16/2025",

"customerId": 2,

"customerName": "Jane Doe",

"product": "Smartphone",

"category": "Electronics",

"quantity": 3,

"unitPrice": 800.00

}

]