Part 1: Solution of SQL Schema Design

1. Relationship Between `TEntryTransaction` and 5 Entry Tables

   The `TEntryTransaction` table is a central table that stores general transaction details. Each row in `TEntryTransaction` has an `entryId` that uniquely identifies an entry type. To capture detailed information specific to each entry type, five specialized tables are created: `BasicBankEntry`, `DistributionInterest`, `Dividend`, `Contribution`, and `Investment`.

   - Foreign Key Relationships: Each specialized table (`BasicBankEntry`, `DistributionInterest`, `Dividend`, `Contribution`, `Investment`) has an `entryId` column that serves as a foreign key referencing the `entryId` in `TEntryTransaction`. This relationship ensures that every entry type in these tables is associated with a transaction in the `TEntryTransaction` table.
   - Unique Index: A unique index, `idx_entryId`, has been added to the `entryId` column of the `TEntryTransaction` table to ensure that each `entryId` is unique across transactions. This is also a basic requirement for MySQL when defining foreign keys.

2. Ensuring Data Integrity and Referential Integrity

   1) Data Integrity:
   - Each specialized entry table has a primary key (`id`) to uniquely identify rows within that table.
   - `entryId` columns in the specialized tables are not allowed to be null (`NOT NULL` constraint), ensuring that every entry in these tables corresponds to a valid transaction.

   2) Referential Integrity:
   - Foreign key constraints ensure that each `entryId` in the specialized tables corresponds to an existing `entryId` in the `TEntryTransaction` table.
   - On Delete Restrict: If a record in `TEntryTransaction` is deleted, the database will prevent the deletion if there are corresponding records in any of the specialized tables, thereby preserving data integrity.

3. Considerations for Future Scalability and Maintainability

   1) Scalability:
   - Additional Entry Types: If new entry types are needed in the future, you can add new specialized tables following the same pattern. Each new type will have its own table, linked to `TEntryTransaction` through the `entryId` foreign key.
   - Performance: Indexing `entryId` ensures efficient queries on the `T_EntryTransaction` table. As the volume of data grows, this will help maintain performance for lookups and joins.

   2) Maintainability:
   - Modular Design: Separating entry types into distinct tables keeps the design modular and manageable. Each entry type table is responsible only for its own data, which simplifies updates and maintenance.

- Code Consistency: By following the pattern of having a unique table for each entry type, the design remains consistent and predictable, making it easier to implement and maintain code that interacts with these tables.

3) Extensibility:

- Future Enhancements: If additional attributes or new entry types are required, you can easily extend the existing tables or add new ones with minimal impact on the existing schema. The foreign key relationships will continue to enforce referential integrity.

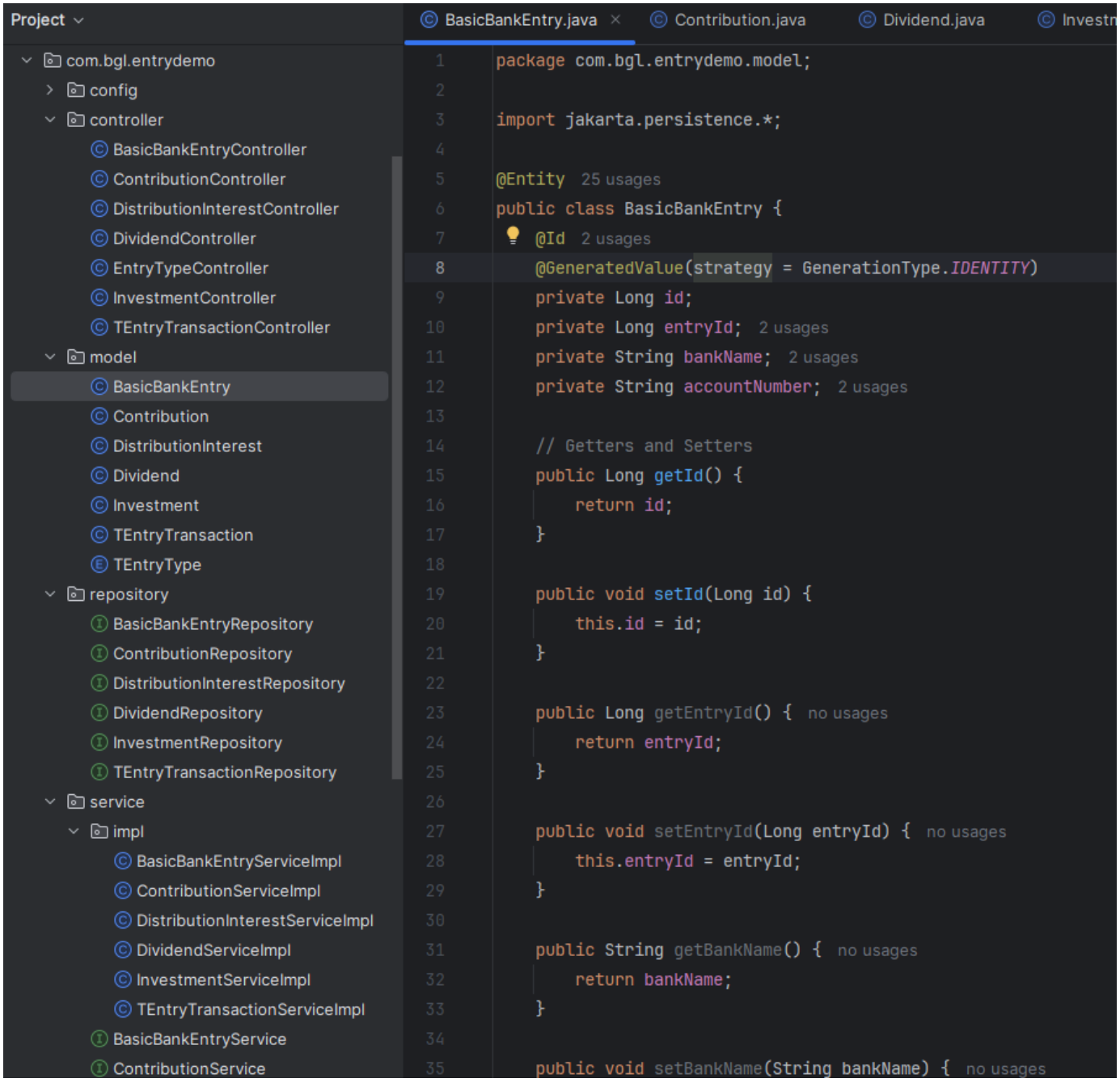
4. Java classes for 5 entries

1) BasicBankEntry:

```
@Entity
public class BasicBankEntry {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private Long entryId;
   private String bankName;
   private String accountNumber;

   // Getters and Setters
   public Long getId() {
      return id;
   }
   ……
}
```

2) DistributionInterest, Dividend, Contribution, Investment classes similar to BasicBankEntry. Please see following class structure in Model folder:


Project
com.bgl.entrydemo
config
controller
BasicBankEntryController
ContributionController
DistributionInterestController
DividendController
EntryTypeController
InvestmentController
TEntryTransactionController
model
BasicBankEntry
Contribution
DistributionInterest
Dividend
Investment
TEntryTransaction
TEntryType
repository
BasicBankEntryRepository
ContributionRepository
DistributionInterestRepository
DividendRepository
InvestmentRepository
TEntryTransactionRepository
service
impl
BasicBankEntryServiceImpl
ContributionServiceImpl
DistributionInterestServiceImpl
DividendServiceImpl
InvestmentServiceImpl
TEntryTransactionServiceImpl
BasicBankEntryService
ContributionService
BasicBankEntry.java
Contribution.java
Dividend.java
package com.bgl.entrydemo.model;
import jakarta.persistence.*;
@Entity 25 usages
public class BasicBankEntry {
@Id 2 usages
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
private Long entryId; 2 usages
private String bankName; 2 usages
private String accountNumber; 2 usages
// Getters and Setters
public Long getId() {
return id;
}
public void setId(Long id) {
this.id = id;
}
public Long getEntryId() { no usages
return entryId;
}
public void setEntryId(Long entryId) { no usages
this.entryId = entryId;
}
public String getBankName() { no usages
return bankName;
}
public void setBankName(String bankName) { no usages

Part 2: Project Development

1. Backend project solution design

   Backend project is designed and implemented with Spring Boot. It supports CRUD operations for the `TEntryTransaction` entity.

   1) Project Setup
   - Dependencies: The Spring Boot project includes dependencies for JPA, MySQL, and Web to handle database interactions and provide REST API functionality.

   2) Entity Creation
   - Entities: Define JPA entities for `TEntryTransaction` and the five entry types. The entities map to database tables and include necessary fields and annotations.
   - Annotations: Use Jakarta EE annotations (`@Entity`, `@Id`, `@GeneratedValue`, `@Column`, etc.) for defining entity mappings and constraints.

   3) Repositories
   - Interfaces: Implement repositories by extending `JpaRepository` for both `TEntryTransaction` and entry types, enabling CRUD operations and custom queries if needed.

   4) REST Controllers
   - Endpoints: Create REST controllers to handle CRUD operations for `TEntryTransaction` and entry types. The controllers include methods for creating, reading, updating, and deleting resources.
   - Mappings: Use `@RestController` and `@RequestMapping` annotations to define RESTful endpoints and handle HTTP requests.

   5) Service Layer
   - Service Interfaces: Define service interfaces for business logic.
   - Service Implementations: Implement service classes that interact with repositories to perform CRUD operations and encapsulate business logic.

2. Frontend project

   Frontend solution implemented with ReactJS framework. It supports CRUD operations for the `TEntryTransaction` entity.

   1) Project Setup
   - Framework: Set up the frontend project using ReactJS, creating a modern, responsive user interface.

   2) Component Structure
   - List Component: Displays a list of `TEntryTransaction` records.
   - Create Component: Form for adding new `TEntryTransaction` records.
   - Update Component: Form for editing existing `TEntryTransaction` records.
   - Delete Component: Confirmation dialog for deleting `TEntryTransaction` records.

- Dialog Component: Custom dialog for confirmation messages.

3) API Integration

- API Calls: Implement API calls using Axios or Fetch API to interact with the Spring Boot backend.
- Create: POST request to add a new `TEntryTransaction`.
- Read: GET request to fetch `TEntryTransaction` records.
- Update: PUT request to update an existing `TEntryTransaction`.
- Delete: DELETE request to remove a `TEntryTransaction`.
- EntryType Dropdown: Fetch dropdown list content for `EntryType` from the backend `TEntryType` class using a dedicated API endpoint.

4) Error Handling and Validation

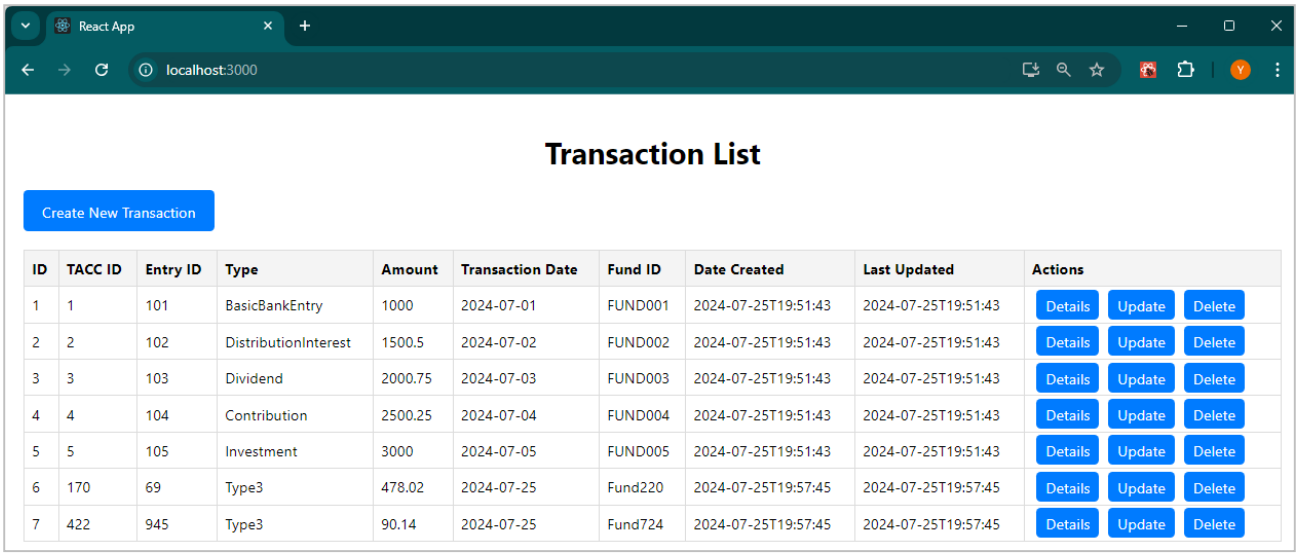
- Form Validation: Implement client-side form validation to ensure data integrity before submitting API requests.
- Error Handling: Handle API errors gracefully, providing user-friendly messages and feedback.

Features summary:

- CRUD Operations: Users can create, read, update, and delete `TEntryTransaction` records through intuitive forms and dialogs.
- List Display: The list component provides a clean and organized view of all `TEntryTransaction` records, with options to edit or delete each entry.
- Custom Confirmation Dialogs: Custom dialogs are used for confirmation messages, enhancing the user experience.
- EntryType Dropdown: Dynamic dropdown list for `EntryType`, ensuring the system can adapt to future changes and expansions easily.

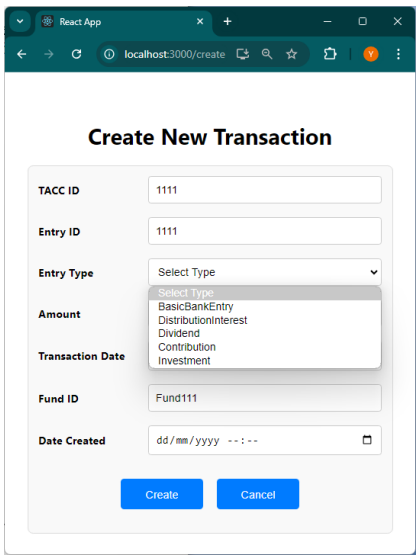
3. Main Functions

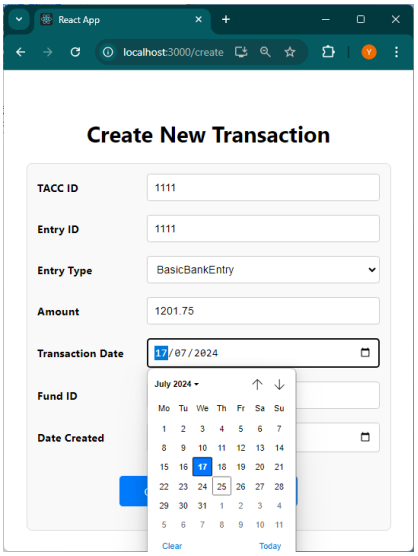
Input http://localhost:3000 in a web browser, Transaction List page will display. All transaction data stored in database will be searched and showed up.

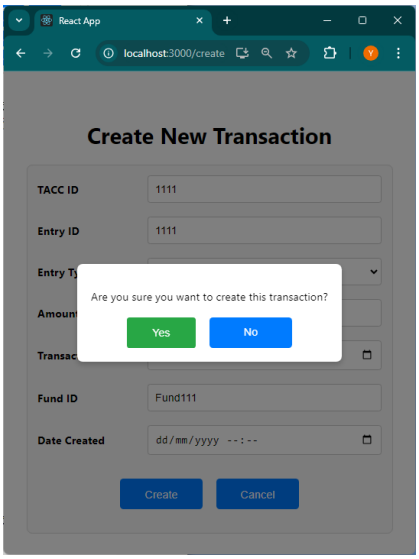


ID	TACC ID	Entry ID	Type	Amount	Transaction Date	Fund ID	Date Created	Last Updated	Actions
1	1	101	BasicBankEntry	1000	2024-07-01	FUND001	2024-07-25T19:51:43	2024-07-25T19:51:43	Details Update Delete
2	2	102	DistributionInterest	1500.5	2024-07-02	FUND002	2024-07-25T19:51:43	2024-07-25T19:51:43	Details Update Delete
3	3	103	Dividend	2000.75	2024-07-03	FUND003	2024-07-25T19:51:43	2024-07-25T19:51:43	Details Update Delete
4	4	104	Contribution	2500.25	2024-07-04	FUND004	2024-07-25T19:51:43	2024-07-25T19:51:43	Details Update Delete
5	5	105	Investment	3000	2024-07-05	FUND005	2024-07-25T19:51:43	2024-07-25T19:51:43	Details Update Delete
6	170	69	Type3	478.02	2024-07-25	Fund220	2024-07-25T19:57:45	2024-07-25T19:57:45	Details Update Delete
7	422	945	Type3	90.14	2024-07-25	Fund724	2024-07-25T19:57:45	2024-07-25T19:57:45	Details Update Delete

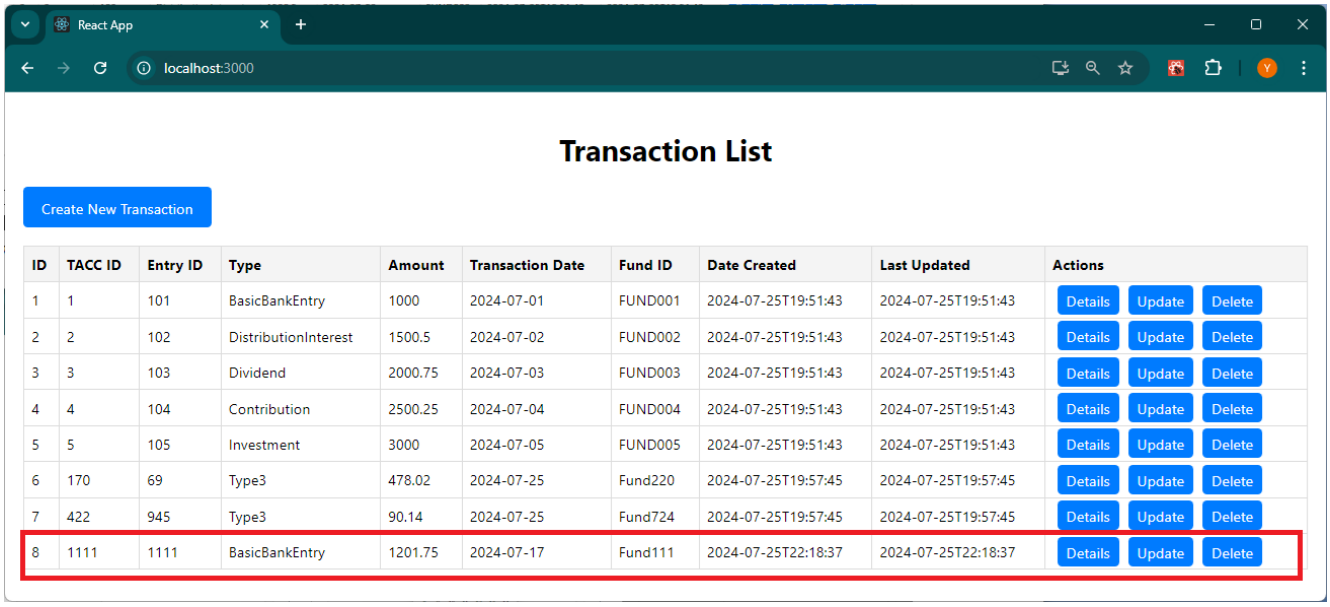
Click **Create New Transaction** button, to create a new transaction. Only number can be input in the **TACC ID** and **Entry ID** component. You can only choose one item in **Entry Type** drop down list, which is retrieved from **EntryType** java class in the backend application service.



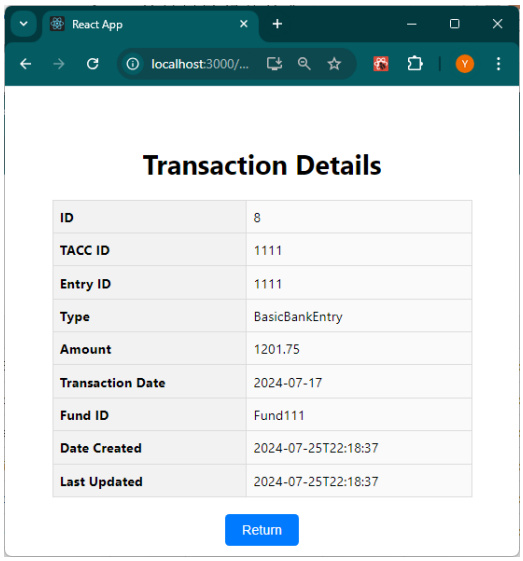




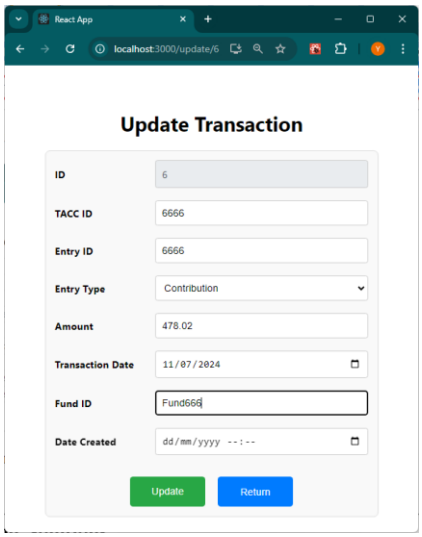
You can input decimal in **Account** item, then choose a **Transaction Date**, input character and number in **Fund ID**, select a create date or leave it blank, then click **Create.**
A confirmation message bog will show up. Click **Yes**, data will be recorded into database, and be shown in **Transaction List** page, as shown below, in which **ID** and **Last Update** time items are added to data records automatically:

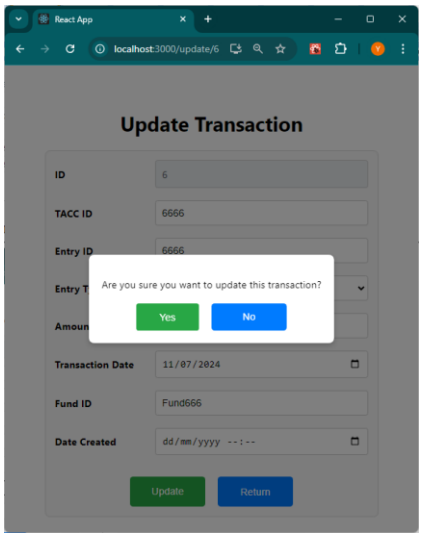


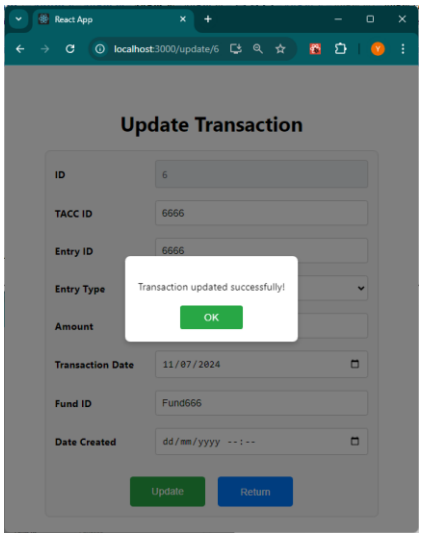
Click **Detail** button in data list to see detail information of the data:



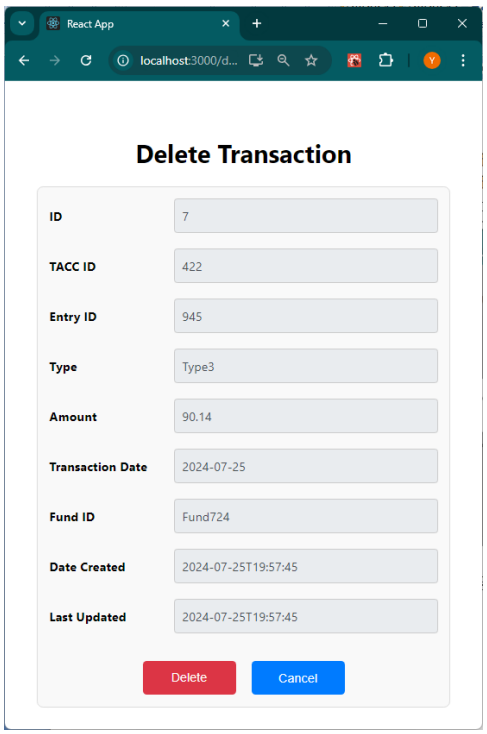
Click **Update** button to modify a data record. After items are updated and click **Update**, and click **Yes** in confirmation dialog box, after data is modified in database, a Success dialog message box will show up. Click **Yes** and **Return** button to return to **Transaction List** page.

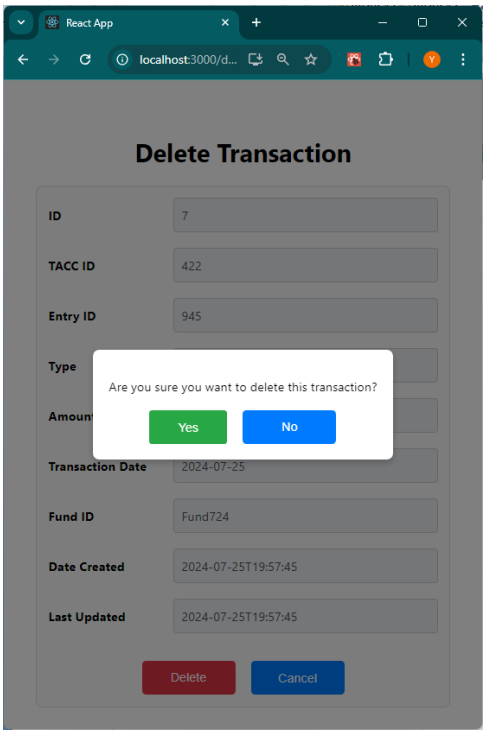




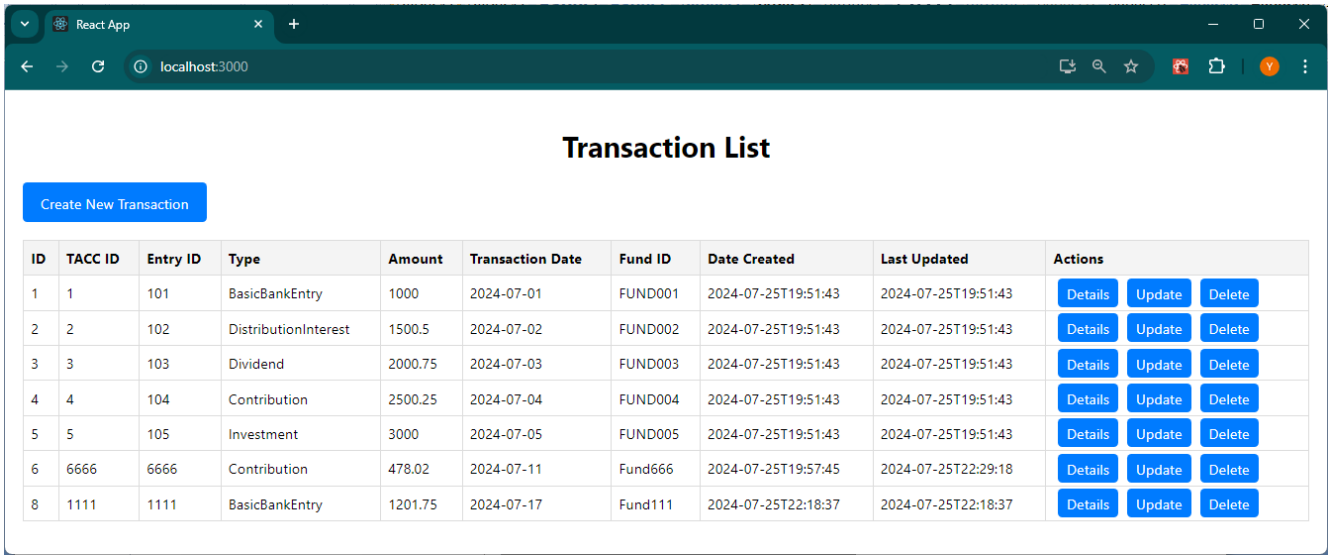


Click **Delete** button in **Transaction List** page, and click **Delete** button in **Delete Transaction** page, click **Yes** in **confirmation dialog box**, to delete a data record.





Data will be deleted in Database, and will not be shown in **Transaction List** page, as shown below (Data with ID 7 is deleted).


React App
localhost:3000
Transaction List
Create New Transaction
ID TACC ID Entry ID Type Amount Transaction Date Fund ID Date Created Last Updated Actions
1 1 101 BasicBankEntry 1000 2024-07-01 FUND001 2024-07-25T19:51:43 2024-07-25T19:51:43 Details Update Delete
2 2 102 DistributionInterest 1500.5 2024-07-02 FUND002 2024-07-25T19:51:43 2024-07-25T19:51:43 Details Update Delete
3 3 103 Dividend 2000.75 2024-07-03 FUND003 2024-07-25T19:51:43 2024-07-25T19:51:43 Details Update Delete
4 4 104 Contribution 2500.25 2024-07-04 FUND004 2024-07-25T19:51:43 2024-07-25T19:51:43 Details Update Delete
5 5 105 Investment 3000 2024-07-05 FUND005 2024-07-25T19:51:43 2024-07-25T19:51:43 Details Update Delete
6 6666 6666 Contribution 478.02 2024-07-11 Fund666 2024-07-25T19:57:45 2024-07-25T22:29:18 Details Update Delete
8 1111 1111 BasicBankEntry 1201.75 2024-07-17 Fund111 2024-07-25T22:18:37 2024-07-25T22:18:37 Details Update Delete

Part 3: Refactoring Solutions

3-1 Create Mockup Data

1. Mock Data Generator program Analysis

   The design of the Mock Data Generator program is focusing on generate large dataset with a high speed. So the program is designed with multithreading, batch processing, and performance enhancement. Program features includs:

   1) Multithreading
   - Thread Pool Usage: The program utilizes a thread pool with a fixed number of threads (`NUM_THREADS`), which is set to 10 in this example. This approach allows concurrent execution of tasks to generate mock data.
   - Task Distribution: It divides the data generation workload among multiple threads. Each thread handles a portion of the data records, improving overall efficiency and reducing the time required for data generation.

   2) Batch Processing
   - Batch Size Management: Data records are inserted into the database in batches of `BATCH_SIZE`, which is set to 5000 records. This reduces the number of database transactions and improves performance by minimizing the overhead associated with each insert operation.
   - Commit and Rollback: The program commits the batch of records to the database and performs a rollback in case of any error, ensuring data integrity and consistency.

   3) Performance Enhancement
   - Concurrent Execution: By leveraging multithreading, the program can generate and insert data in parallel, significantly speeding up the overall execution time compared to a single-threaded approach.
   - Efficient Resource Utilization: The use of a fixed thread pool allows efficient management of system resources, ensuring that threads are reused and avoiding the overhead of frequently creating and destroying threads.
   - Batch Execution: Executing batches of SQL statements reduces the number of database round-trips, which improves performance by minimizing the interaction time with the database.

   4) Error Handling
   - Exception Handling: The program includes error handling mechanisms to catch and print SQL exceptions. In case of an error, it performs a rollback to maintain data integrity.

3-2 Refactor Solution

1. Database Schema Changes Analysis

1) Database Schema Changes

- The `TAccountTags` table will be altered to remove the `year` column.
- Backup procedures will be carefully planned, especially when backing up related tables due to foreign key constraints.
- Testing should be conducted before the actual migration to identify any performance bottlenecks.

SQL Script to Modify Schema:

ALTER TABLE TAccountTags DROP COLUMN year;

2) Data Migration Strategy

- Create a Backup Table: Create a backup table with the same structure as TAccountTags, but without the auto-increment constraint on the id field.
- Copy Data: Copy data from the original TAccountTags table to the backup table, ensuring id values remain the same.
- Restore Auto-Increment: If necessary, re-enable the auto-increment on the id field in the backup table.

2. Operation Steps:

1) Backup Data (example):

(1) Create a Backup Table:

```
CREATE TABLE TAccountTags_backup (
    id BIGINT NOT NULL,
    name VARCHAR(255) DEFAULT NULL,
    color VARCHAR(20) DEFAULT NULL,
    type VARCHAR(50) DEFAULT NULL,
    year SMALLINT DEFAULT NULL,
    isSystemLabel BIT(1) DEFAULT NULL,
    code VARCHAR(100) DEFAULT NULL,
    item_order INT DEFAULT NULL,
    PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

(2) Copy Data

```
INSERT INTO `TAccountTags_Backup` (id, name, color, type, isSystemLabel, code,
item_order)
SELECT id, name, color, type, isSystemLabel, code, item_order
FROM `TAccountTags`;
```

(3) Enable Auto-Increment

```
ALTER TABLE `TAccountTags_Backup` MODIFY `id` bigint NOT NULL
AUTO_INCREMENT;
```

2) Alter Table:

ALTER TABLE TAccountTags DROP COLUMN year;

3) Code Changes:

Remove references to the `year` attribute in the entity, service, repository, and controller layers.

3. Code Refactoring (Java Code Changes)

1) Entity Class Update (Example):

```
@Entity
public class TAccountTags {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String color;
    private String type;
    private Boolean isSystemLabel;
    private String code;
    private Integer itemOrder;
}
```

2) Service and Repository Layer:

- Ensure all CRUD operations in the service and repository layers do not refer to the `year` attribute.
- Update any business logic that depends on the `year` attribute.

3) Controller Layer:

- Adjust endpoints and request/response DTOs to exclude the `year` attribute.

4) Frontend Layer:

- **Form Adjustments**: Remove any form fields or UI components related to the year attribute.
- **API Adjustments**: Ensure the frontend no longer expects the year attribute in the API responses for TAccountTags.

4. Performance Considerations

To ensure the refactored system performs efficiently with a large dataset, such as the one with around 1 billion records in the `TEntryTransaction` table, the following performance considerations and strategies should be implemented (if applicable):

1) Indexing

- Create Appropriate Indexes: Ensure that indexes are in place for columns that are frequently used in queries, especially those involved in WHERE clauses, ORDER BY operations. For example, indexes on `taccId`, `entryId` in `TEntryTransaction` can significantly enhance query performance.
- Use Composite Indexes: If queries involve multiple columns, consider creating composite indexes. For example, an index on `(taccId, entryId)` might be beneficial if these columns are often queried together.

2) Partitioning

- Partition Large Tables: Partitioning the `TEntryTransaction` table can help manage large datasets efficiently. For instance, partition the table by date or by `entryId` (enrtyType). This allows the database to quickly locate and access relevant partitions rather than scanning the entire table.

  A example:

```
ALTER TABLE TEntryTransaction
PARTITION BY RANGE (YEAR(transactionDate)) (
    PARTITION p0 VALUES LESS THAN (1990),
    PARTITION p1 VALUES LESS THAN (2000),
    PARTITION p2 VALUES LESS THAN (2010),
    PARTITION p3 VALUES LESS THAN (2020),
    PARTITION p4 VALUES LESS THAN (MAXVALUE)
 );
```

3) Batch Processing

- Batch Data Migration: During data migration or when processing large volumes of data, use batch processing techniques to handle data in chunks. This reduces the load on the system and avoids locking large amounts of data at once.

  A example:

```
final int batchSize = 1000;
for (int i = 0; i < data.size(); i++) {
    processRecord(data.get(i));
    if (i % batchSize == 0) {
        commitBatch();
    }
}
```

4) Query Optimization

- Analyze and Optimize Queries: Regularly analyze query performance using tools like MySQL's `EXPLAIN` command to understand how queries are executed and identify any potential bottlenecks. Optimize queries by rewriting them for better performance or adjusting indexes.

  A example:

EXPLAIN SELECT * FROM TEntryTransaction WHERE taccId = ? AND transactionDate BETWEEN ? AND ?;

5) Caching

- Implement Caching: Use caching mechanisms to store frequently accessed data in memory, reducing the need for repetitive database queries. Tools like Redis or Memcached can be integrated to cache query results or frequently accessed data.

  A example:

```
String cacheKey = "TEntryTransaction_" + entryId;
TEntryTransaction entry = cache.get(cacheKey);
if (entry == null) {
    entry = database.findById(entryId);
    cache.put(cacheKey, entry);
}
```

6) Testing

- Load Testing: Perform load testing to simulate large-scale data access and ensure the system can handle the expected volume of queries and transactions without performance degradation.

5. Deploy Changes:
   - Use a rolling deployment strategy to ensure minimal downtime.

6. Test Thoroughly:
   - Perform comprehensive testing to ensure the application works as expected after the changes.

7. Solution Summary
   - Schema Change: Dropping the `year` column.
   - Data Migration: Backup and modify schema.
   - Code Refactor: Update Java code to exclude `year`, including backend and frontend functions.
   - Performance: Efficiently handle large datasets.

Appendix: Data Backup Constraints and Consideration (If Necessary)

Because there are some constraints between table TAccount, TAccountTags and TAccountTag, operation orders should be followed as following if all the data above will be backup:

1. Data Insertion Order:

- Backup and insert into `TAccount` first because `TAccountTag` depends on it.
- Backup and insert into `TAccountTags` next, also because `TAccountTag` depends on it.
- Backup and insert into `TAccountTag` last, ensuring it references existing `TAccount` and `TAccountTags`.

2. SQL for Data Insertion (Example):

- Insert into TAccount

Firstly: Data with no pid (parent id):
INSERT INTO TAccount (accountID, pid, code, name, accountClass, accountType) VALUES
(1, NULL, 'ACC001', 'Account 1', 'A', 'Type A'),
(2, NULL, 'ACC002', 'Account 2', 'A', 'Type B'),
…

Secondly: Data with pid (parent id):
INSERT INTO TAccount (accountID, pid, code, name, accountClass, accountType) VALUES
(6, 1, 'ACC006', 'Account 6', 'C', 'Type C'),
(7, 1, 'ACC007', 'Account 7', 'A', 'Type A'),
(8, 2, 'ACC008', 'Account 8', 'B', 'Type B'),
(9, 2, 'ACC009', 'Account 9', 'C', 'Type A'),
…

- Insert into TAccountTags (Example)

INSERT INTO TAccountTags (name, color, type, isSystemLabel, code, itemOrder) VALUES
('Tag 1', 'Red', 'Type 1', 2021, 0, 'TAG001', 1),
('Tag 2', 'Blue', 'Type 2', 2022, 0, 'TAG002', 2),
('Tag 3', 'Green', 'Type 1', 2023, 1, 'TAG003', 3),
('Tag 4', 'Yellow', 'Type 3', 2021, 1, 'TAG004', 4),

- Insert into TAccountTag (Example)

INSERT INTO TAccountTag (tagID, taccountID) VALUES
(1, 1), -- Tag 1 for Account 1
(2, 1), -- Tag 2 for Account 1
(3, 2), -- Tag 3 for Account 2
(4, 2), -- Tag 4 for Account 2
(5, 3), -- Tag 5 for Account 3