# **Bug Tracking System**

# Software Requirement Document

## Layered architecture

Models:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **User** | | | | | |
|  |  |  | getter | setter | compare param |
| userId | Private | String | Y | Y | Y |
| userName | Private | String | Y | Y |  |
| userEmail | Private | String | Y | Y |  |
| userType | Protected | UserTypeEnum | Y | Y |  |

## Learnings

**PreparedStatement Methods**

In Java, `PreparedStatement` is an interface in the `java.sql` package that extends the `Statement` interface. It is used to execute parameterized SQL queries, which helps prevent SQL injection attacks and improves performance by precompiling the SQL statement. `PreparedStatement` is particularly useful when you need to execute the same SQL statement with different sets of parameters multiple times.

Here are some commonly used methods of the `PreparedStatement` interface in Java:

**1. Setting Parameters:**

* setString(int parameterIndex, String x): Sets the designated parameter to the given Java `String` value.
* setInt(int parameterIndex, int x): Sets the designated parameter to the given Java `int` value.
* setDouble(int parameterIndex, double x): Sets the designated parameter to the given Java `double` value.
* setDate(int parameterIndex, Date x): Sets the designated parameter to the given Java `Date` value.

**Example:**

PreparedStatement preparedStatement = connection.prepareStatement("INSERT INTO table\_name (column1, column2) VALUES (?, ?)");

preparedStatement.setString(1, "value1");

preparedStatement.setInt(2, 42);

**2. Executing Statements:**

* execute(): Executes the SQL statement.
* executeQuery(): Executes a SQL SELECT query and returns a `ResultSet` object.
* executeUpdate(): Executes a SQL INSERT, UPDATE, or DELETE statement and returns the number of affected rows.

**Example:**

int rowsAffected = preparedStatement.executeUpdate();

**3. Batch Processing:**

* addBatch(): Adds a set of parameters to the batch.
* executeBatch(): Executes the batch of commands.

**Example:**

preparedStatement.addBatch();

int[] rowsAffected = preparedStatement.executeBatch();

**4. Handling Generated Keys:**

* getGeneratedKeys(): Retrieves any auto-generated keys created as a result of executing this `PreparedStatement` object.

**Example:**

ResultSet generatedKeys = preparedStatement.getGeneratedKeys();

These methods provide a convenient and secure way to work with parameterized SQL statements in Java, allowing you to create more flexible and efficient database interactions.

**ResultSet Methods**

In Java, a `ResultSet` is an interface in the `java.sql` package that provides methods for retrieving and manipulating the results of a database query. It represents a set of data returned by a database query, typically a SELECT statement. The data is organized in rows and columns, similar to a table.

Here are some key aspects and methods associated with the `ResultSet` interface:

**1. Navigation:**

* next(): Moves the cursor to the next row in the `ResultSet`. It returns `true` if there is a next row, and `false` if there are no more rows.

**Example:**

ResultSet resultSet = statement.executeQuery("SELECT \* FROM table\_name");

while (resultSet.next()) {

// Process each row

}

**2. Retrieving Data:**

* getString(int columnIndex) / getString(String columnLabel): Retrieves the value of the specified column as a `String`.
* getInt(int columnIndex) / getInt(String columnLabel): Retrieves the value of the specified column as an `int`.
* getDouble(int columnIndex) / getDouble(String columnLabel): Retrieves the value of the specified column as a `double`.

**Example:**

String name = resultSet.getString("name");

int age = resultSet.getInt("age");

**3. Positioning the Cursor:**

* absolute(int row): Moves the cursor to the specified row number.
* relative(int rows): Moves the cursor a relative number of rows, either forward or backward.

**4. Metadata:**

* getMetaData(): Retrieves the `ResultSetMetaData` object, which contains information about the columns in the `ResultSet`.

**Example:**

ResultSetMetaData metaData = resultSet.getMetaData();

int columnCount = metaData.getColumnCount();

**5. Closing:**

* close(): Closes the `ResultSet` object. It is important to close the `ResultSet` when you are done with it to release resources.

**Example:**

resultSet.close();

The `ResultSet` interface provides a flexible and powerful way to interact with the results of a database query in a Java program. It allows you to iterate through the rows, retrieve data from columns, and perform various operations on the data retrieved from the database.

**DataBase design patterns of Parent – Child relationship (Inheritance)**

In the context of database design, when you have a parent class and a child class (representing an inheritance relationship), there are several approaches to modeling this in a relational database. Two common strategies are:

**Single Table Inheritance (STI):**

**Table Structure:** In STI, both the parent and child classes are represented by a single table. The table includes columns for all properties of both the parent and child classes. The child class columns that are specific to the child (and not present in the parent) are allowed to be nullable.

**Advantages:** Simplifies queries and joins as all data is in one table.

**Disadvantages:** Can result in a lot of NULL values for child-specific columns, and may lead to data redundancy.

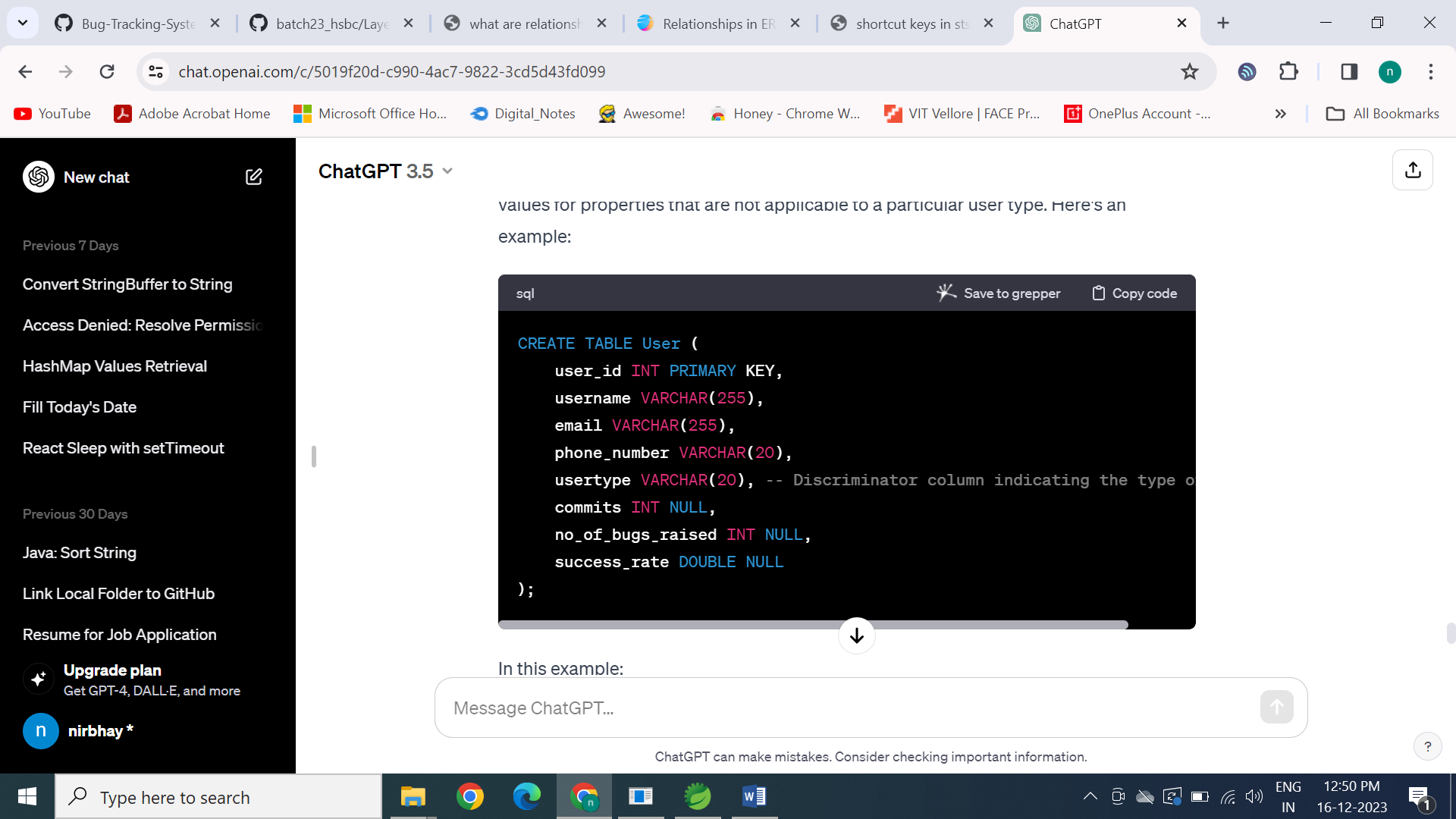
**Class Table Inheritance (CTI):**

**Table Structure:** In CTI, each class (both parent and child) is represented by its own table. The parent table contains common properties, and the child table contains only properties specific to the child class. There is usually a foreign key relationship between the parent and child tables.

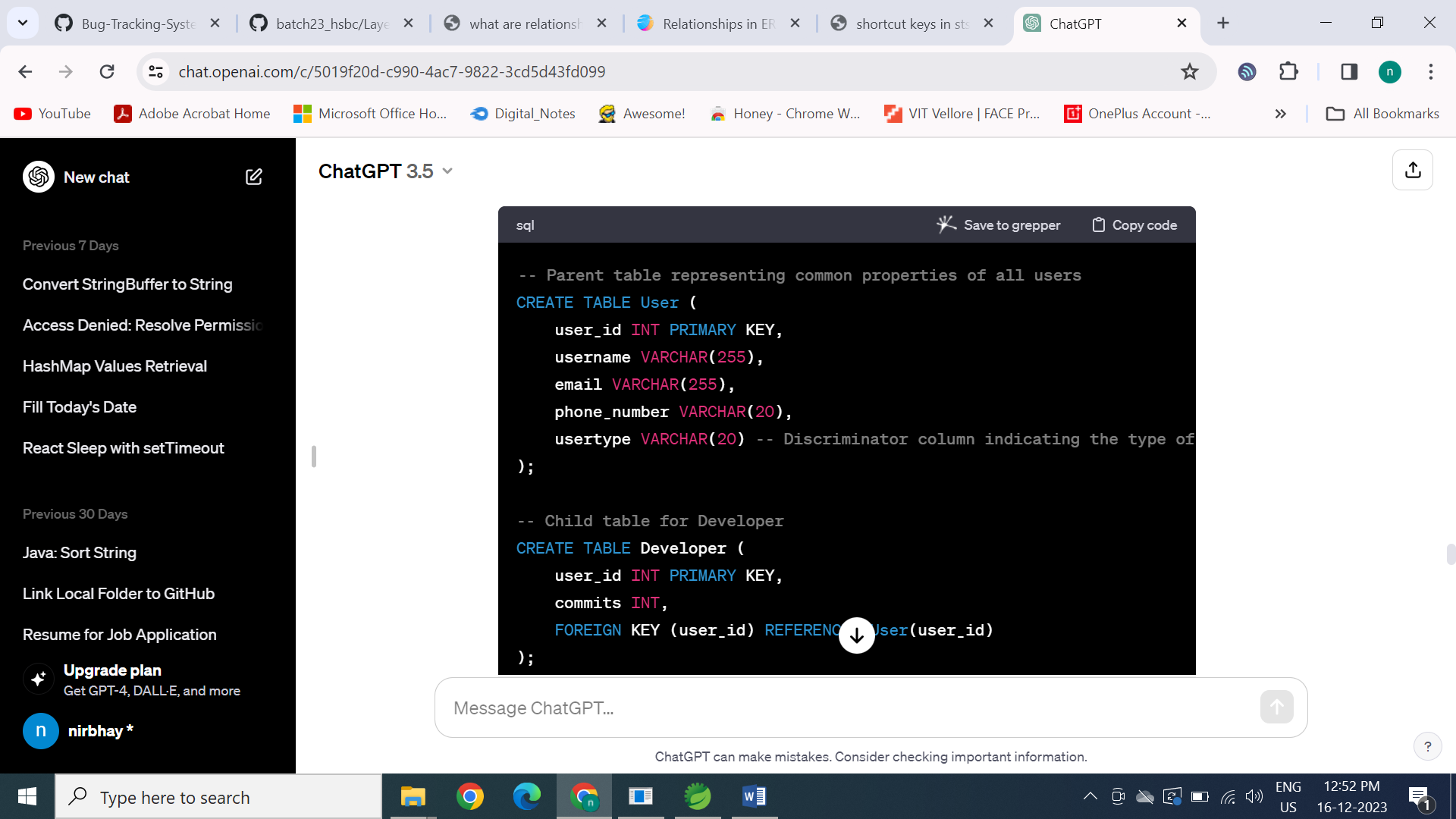
**Advantages:** Reduces data redundancy and avoids NULL values for child-specific columns.

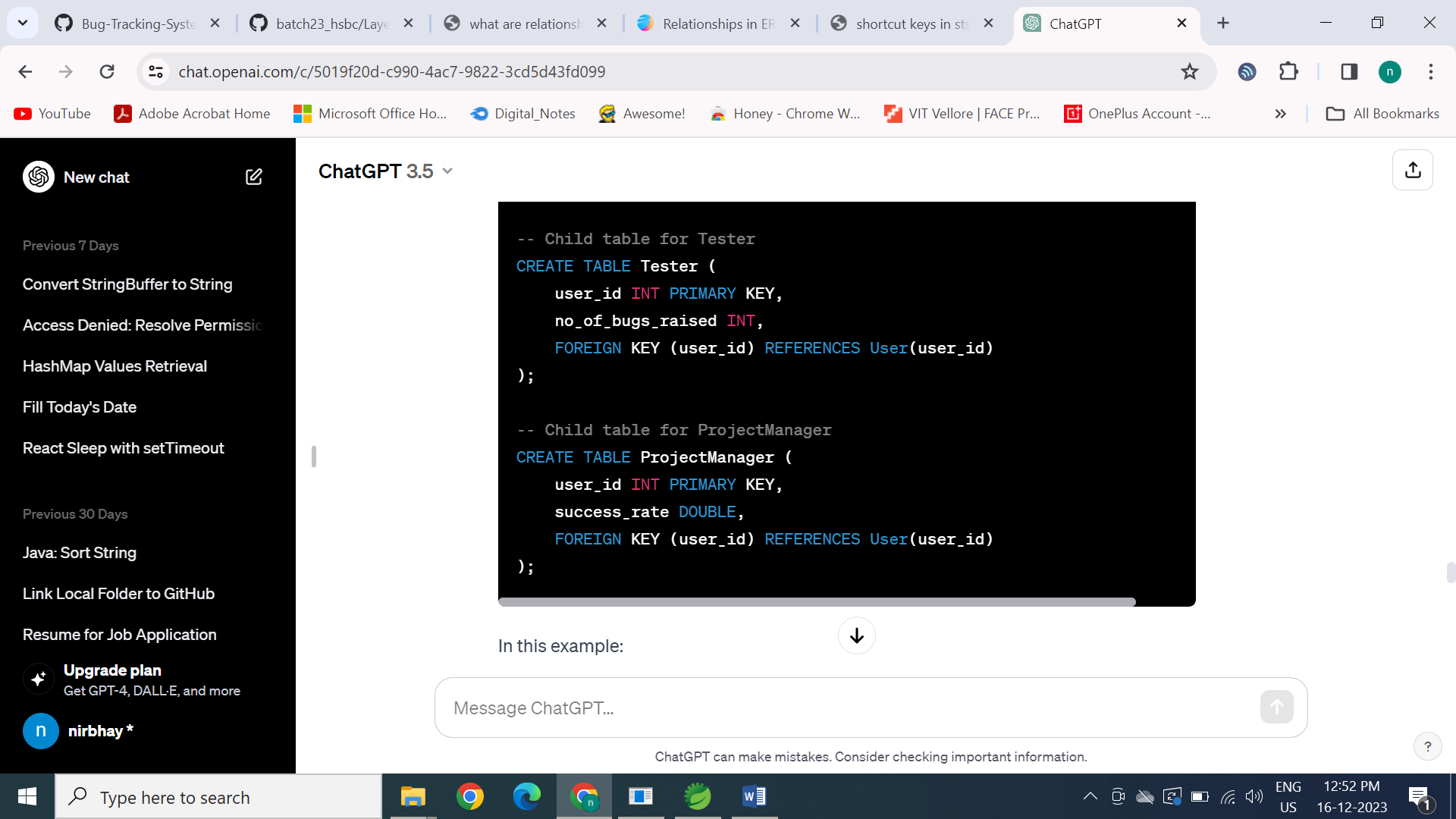
**Disadvantages:** Queries and joins might be more complex, and maintaining referential integrity can be more challenging.

**Example – STI**



**Example – CTI**





-- Insert a Developer

INSERT INTO User (user\_id, username, email, phone\_number, usertype) VALUES (1, 'dev\_user', 'dev@example.com', '1234567890', 'Developer');

INSERT INTO Developer (user\_id, commits) VALUES (1, 100);

-- Insert a Tester

INSERT INTO User (user\_id, username, email, phone\_number, usertype) VALUES (2, 'tester\_user', 'tester@example.com', '9876543210', 'Tester');

INSERT INTO Tester (user\_id, no\_of\_bugs\_raised) VALUES (2, 50);

-- Insert a ProjectManager

INSERT INTO User (user\_id, username, email, phone\_number, usertype) VALUES (3, 'pm\_user', 'pm@example.com', '5555555555', 'ProjectManager');

INSERT INTO ProjectManager (user\_id, success\_rate) VALUES (3, 0.95);