

Module 2 Capstone - TEnmo

Congratulations—you've landed a job with TEnmo, whose product is an online payment service for transferring "TE bucks" between friends. However, they don't have a product yet. Parts of the project have been started, you have been tasked with completing the remaining features using the partial RESTful API server and command-line application given.

Review Franks Comments for the Module 2 Capstone document

The required use case code for the client side has already been completed. There is nothing for your do in it.

You will be working on the server side only implementing the data access and server side APIs

Use Postman to test your server side code

You may run the client to test your server side code *AFTER* you have verified it is correct using Postman

Use cases

Required use cases

You should attempt to complete all of the following required use cases.

1. **[COMPLETE]** As a user of the system, I need to be able to register myself with a username and password.
 1. A new registered user starts with an initial balance of 1,000 TE Bucks.
 2. The ability to register has been provided in your starter code.
2. **[COMPLETE]** As a user of the system, I need to be able to log in using my registered username & password.
 1. Logging in returns an Authentication Token. I need to include this token with all my subsequent interactions with the system outside of registering and logging in.
 2. The ability to log in has been provided in your starter code.
3. As an authenticated user of the system, I need to be able to see my Account Balance.
4. As an authenticated user of the system, I need to be able to *send* a transfer of a specific amount of TE Bucks to a registered user.
 1. I should be able to choose from a list of users to send TE Bucks to.
 2. I must not be allowed to send money to myself.
 3. A transfer includes the User IDs of the from and to users and the amount of TE Bucks.
 4. The receiver's account balance is increased by the amount of the transfer.
 5. The sender's account balance is decreased by the amount of the transfer.
 6. I can't send more TE Bucks than I have in my account.
 7. I can't send a zero or negative amount.
 8. A Sending Transfer has an initial status of *Approved*.
5. As an authenticated user of the system, I need to be able to see transfers I have sent or received.
6. As an authenticated user of the system, I need to be able to retrieve the details of any transfer based upon the transfer ID.

Optional use cases - Not included in tenmo-client - you must write both server and client code for these

If you complete all of the required use cases and are looking for additional challenge, complete as many of the following optional use cases as you can.

7. As an authenticated user of the system, I need to be able to *request* a transfer of a specific amount of TE Bucks from another registered user.
 1. I should be able to choose from a list of users to request TE Bucks from.
 2. I must not be allowed to request money from myself.
 3. I can't request a zero or negative amount.
 4. A transfer includes the User IDs of the from and to users and the amount of TE Bucks.
 5. A Request Transfer has an initial status of *Pending*.
 6. No account balance changes until the request is approved.
 7. The transfer request should appear in both users' list of transfers (use case #5).
8. As an authenticated user of the system, I need to be able to see my *Pending* transfers.
9. As an authenticated user of the system, I need to be able to either approve or reject a Request Transfer.
 1. I can't "approve" a given Request Transfer for more TE Bucks than I have in my account.
 2. The Request Transfer status is *Approved* if I approve, or *Rejected* if I reject the request.
 3. If the transfer is approved, the requester's account balance is increased by the amount of the request.
 4. If the transfer is approved, the requestee's account balance is decreased by the amount of the request.
 5. If the transfer is rejected, no account balance changes.

Sample screens

Feel free to customize, but information displayed and user interaction must be similar

Use case 3: Current balance

```
-----  
Account information for username (User Id: userid)  
  Account #: 2001  
  Balance: 1000.00  
-----
```

Use case 4: Send TE Bucks

```
-----  
Users  
ID          Name  
-----  
313         Bernice  
54          Larry  
-----
```

```
Enter ID of user you are sending to (0 to cancel):
Enter amount:
```

Use case 5: View transfers

```
-----
Transfers
ID          From/To          Amount
-----
23          From: Bernice      $ 903.14
79          To:    Larry          $  12.55
-----
Please enter transfer ID to view details (0 to cancel): "
```

Use case 6: Transfer details

```
-----
Transfer Details
-----
Id: 23
From: Bernice
To: Me Myselfandi
Type: Send
Status: Approved
Amount: $903.14
```

Use case 7: Requesting TE Bucks

```
-----
Users
ID          Name
-----
313         Bernice
54          Larry
-----

Enter ID of user you are requesting from (0 to cancel):
Enter amount:
```

Use case 8: Pending requests

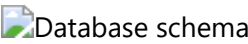
```
-----
Pending Transfers
ID          To          Amount
```

```
-----
88          Bernice          $ 142.56
147         Larry           $  10.17
-----
Please enter transfer ID to approve/reject (0 to cancel): "
```

Use case 9: Approve or reject pending transfer

```
1: Approve
2: Reject
0: Don't approve or reject
-----
Please choose an option:
```

Database schema



tenmo_user table

Stores the login information for users of the system.

Field	Description
user_id	Unique identifier of the user
username	String that identifies the name of the user; used as part of the login process
password_hash	Hashed version of the user's password
role	Name of the user's role

account table

Stores the accounts of users in the system.

Field	Description
account_id	Unique identifier of the account
user_id	Foreign key to the users table; identifies user who owns account
balance	The amount of TE bucks currently in the account
created_at	When account was created; automatically set by data source; not used in the application

transfer_type table

Stores the types of transfers that are possible.

Field	Description
-------	-------------

Field	Description
transfer_type_id	Unique identifier of the transfer type
transfer_type_desc	String description of the transfer type

There are two types of transfers:

transfer_type_id	transfer_type_desc	Purpose
1	Request	Identifies transfer where a user requests money from another user
2	Send	Identifies transfer where a user sends money to another user

[transfer_status](#) table

Stores the statuses of transfers that are possible.

Field	Description
transfer_status_id	Unique identifier of the transfer status
transfer_status_desc	String description of the transfer status

There are three statuses of transfers:

transfer_status_id	transfer_status_desc	Purpose
1	Pending	Identifies transfer that hasn't occurred yet and requires approval from the other user
2	Approved	Identifies transfer that has been approved and occurred
3	Rejected	Identifies transfer that wasn't approved

[transfer](#) table

Stores the transfers of TE bucks.

Field	Description
transfer_id	Unique identifier of the transfer
transfer_type_id	Foreign key to the transfer_types table; identifies type of transfer
transfer_status_id	Foreign key to the transfer_statuses table; identifies status of transfer
account_from	Foreign key to the accounts table; identifies the account that the funds are being taken from

Field	Description
account_to	Foreign key to the accounts table; identifies the account that the funds are going to
amount	Amount of the transfer
when_recorded	When transfer was created; automatically set by data source ; not used in the application

Note: there are two check constraints in the DDL that creates the [transfer](#) table. Be sure to take a look at [tenmo.sql](#) to understand these constraints.

How to set up the database

Create a new Postgres database called [tenmo](#). Run the [database/tenmo.sql](#) script in pgAdmin to set up the database.

Datasource

A Datasource has been configured for you in [/src/resources/application.properties](#).

```
# datasource connection properties
spring.datasource.url=jdbc:postgresql://localhost:5432/tenmo
spring.datasource.name=tenmo
spring.datasource.username=postgres
spring.datasource.password=postgres1
```

JdbcTemplate

If you look in [/src/main/java/com/techelevator/dao](#), you'll see [JdbcUserDao](#). This is an example of how to get an instance of [JdbcTemplate](#) in your DAOs. If you declare a field of type [JdbcTemplate](#) and add it as an argument to the constructor, Spring automatically injects an instance for you:

```
@Service
public class JdbcUserDao implements UserDao {

    private JdbcTemplate jdbcTemplate;

    public JdbcUserDao(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

Testing

DAO integration tests

`com.techelevator.dao.BaseDaoTests` has been provided for you to use as a base class for any DAO integration test. It initializes a Datasource for testing and manages rollback of database changes between tests.

`com.techelevator.dao.JdbUserDaoTests` has been provided for you as an example for writing your own DAO integration tests.

Remember that when testing, you're using a copy of the real database. The schema and data for the test database are defined in `/src/test/resources/test-data.sql`. The schema in this file matches the schema defined in `database/tenmo.sql`.

Authentication / Finding a Logged In User Information

The user registration and authentication functionality for the system has already been implemented. If you review the login code, you'll notice that after successful authentication, an instance of `AuthenticatedUser` is stored in the `currentUser` member variable of `App`. The user's authorization token—meaning JWT—can be accessed from `App` as `currentUser.getToken()`.

When the use cases refer to an "authenticated user", this means a request that includes the token as a header. You can also reference other information about the current user by using the `User` object retrieved from `currentUser.getUser()`.

Review the code given in the client-side `ConsoleService`, `displayAnAccount()` method for examples on use of the `currentUser` object to obtain information about a user.

Things to consider before you start working on your project

1. Start the server provided to be sure it starts without error.
2. Start the client provided and register at least one user.
3. Use PGAdmin to verify that an account and user has been successfully created in the database.
4. Use PostMan to test the endpoints provided for the Tenmo app:
 1. `/account` - should return all accounts in the database
 2. `/account?userid=userid` - (example: `/account?userid=1001`) - should return all accounts for the userid 1001
 3. `/account/accountId` - (example: `/account/2001`) - should return the account 2001
 4. `/user` - should return all registered users
 5. `/user?userid=userid` - (example: `/user?userid=1001`) - should return user with userid 1001
5. Optional: Update the balance in an account using PostMan:
 1. Set the request type to `PUT`
 2. Set the URL to: `http://localhost:8080/account`
 3. Add/Check for a Header: `Content-Type application/json`
 4. Add `raw` JSON in the Body of the request with the required values. The values shown below are for the first account added to the database. Choose any value you'd like (that is not the current balance):

```
{
  "account_id" : 2001,
  "user_id"    : 1001,
  "balance"    : 31952
}
```

5. [Send](#) the request to the server.
 6. You should receive a JSON object for the account showing the changed balance.
 7. If you get a status code [200](#) and nothing returned, be sure the account you specified in the JSON exists then check the messages on the server side for an error.
 8. If you set a status code other than [200](#), check the messages on the server side for an error.
6. Sample JSON for testing your Transfer controller processing from Postman:

```
{
  "transferId": 99999,
  "transferStatus": "APPROVED",
  "transferType": "SEND",
  "fromTenmoAccount": {
    "account_id" : "2001",
    "user_id"    : "1001",
    "balance"    : "8888.88"
  },
  "toTenmoAccount": {
    "account_id" : "2002",
    "user_id"    : "1002",
    "balance"    : "9999.99"
  },
  "amount": "12345.67"
}
```

7. Review the [Module-2 Capstone Design.pdf](#) and be sure you understand the overall design of the system as well as the purpose of each component and what they interact with.

Also be sure you understand the project structure of the client-side and server-side projects provided and the purpose/use of each folder, package and class given in each project.
8. Review all the code given related to the Tenmo Application in the client packages ([model](#), [ConsoleService](#), [TenmoService](#)) and the [App](#) class. Code in these may be used a model for new features you will be implementing.
9. Use IntelliJ to find the [TODOs](#) in the client code [TenmoService](#). It will help you focus on what to do next. If you don't know how to do this, ask your instructor for assistance. Note: Although the [TODOs](#) are in the client code [TenmoService](#), some of them may require coding on the server-side as well.
10. Review all the code given related to the Tenmo Application and in the server packages ([controller.TenmoController](#), [datasource](#)). Code in these may be used a model for new features you will be implementing.

11. Remember: **Analyze, Design, Code, Refactor**
12. Follow the **Happy Path** first. Don't worry about edge cases until your **Happy Paths** are done.
13. **Don't overthink the problem.** If you have designed a viable solution, code and test it before you decide it's wrong or needs tweaking. That's what **Refactor** is for. It is much easier to debug code that is written than trying to debug code as you write it.

Smile when you code!