CS673 Software Engineering
Team 1 - Trackr
Software Design Document

Team Member	Role(s)	Signature	Date
Timothy Flucker	Design / Implementation Leader	*Timothy Flucker*	05/27/2022
Jean Dorancy	Team Leader	*Jean Dorancy*	05/29/2022
Xiaobing Hou	Security Leader	*Xiaobing Hou*	05/29/2022
Weijie Liang	QA Leader	*Weijie Liang*	05/29/2022

Revision history

Version	**Author**	**Date**	**Change**
0.1	**Timothy Flucker**	**05/27/2022**	**Initial content**
0.2	**Jean Dorancy**	**05/29/2022**	**UI Mocks and React Container Pattern**
1.0	**Timothy Flucker**	**05/30/2022**	**Minor updates, prepare to release for Iteration 1**
1.1	**Timothy Flucker**	**06/03/2022**	**Iteration 2- Updating package structure + description**
1.2	**Timothy Flucker**	**06/15/2022**	**Iteration 3 - Final Updates**

Introduction

Software Architecture

Class Diagram

UI Design (if applicable) pending convo with Professor

Database Design (if applicable)

Security Design

Business Logic and/or Key Algorithms

Design Patterns

Any Additional Topics you would like to include.

References

Glossary

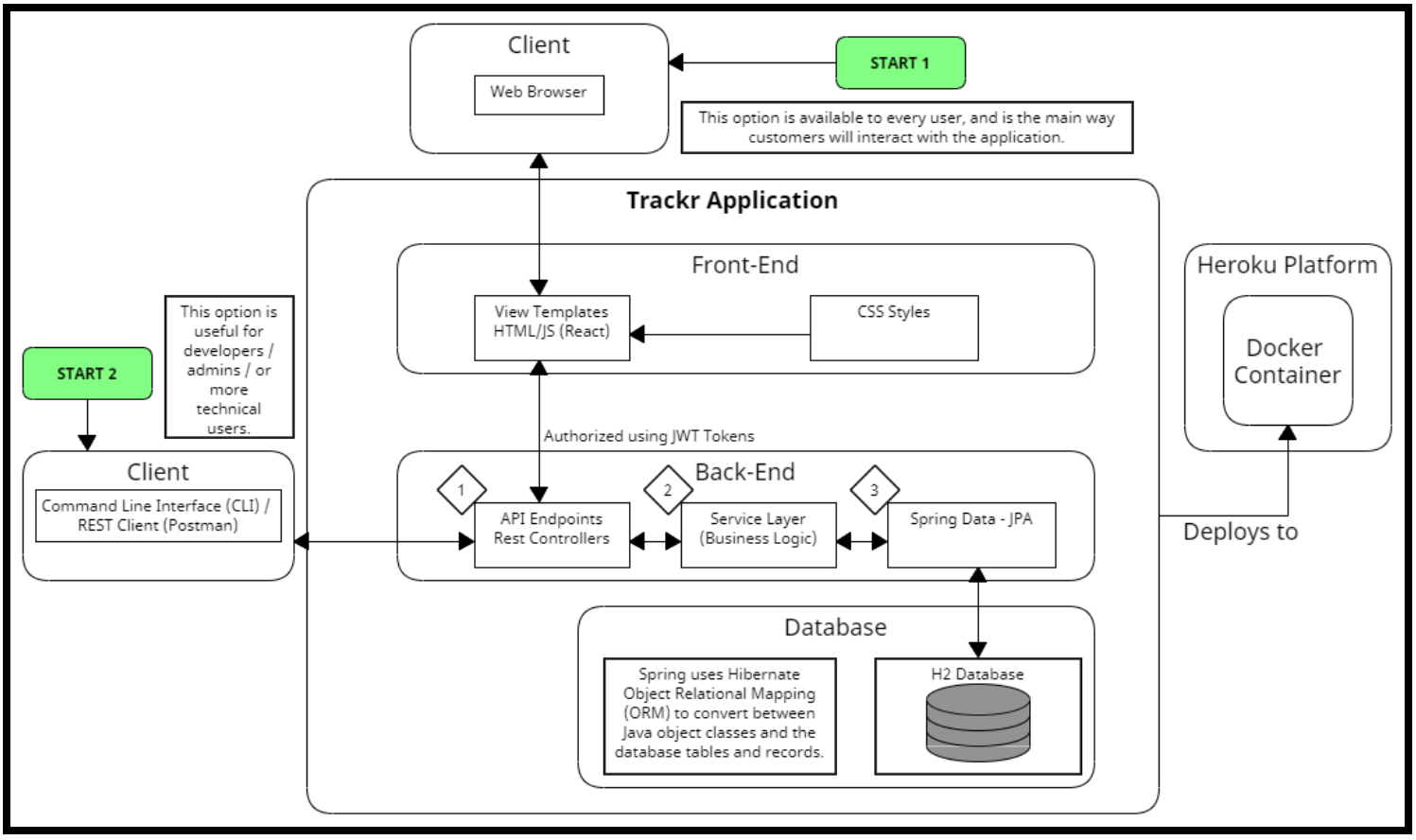
- # Introduction

  This document is meant to define and provide detail for the design and implementation of our group project, Trackr. This application allows users to view financial transactions against their bank accounts so that they can understand their overall financial well-being and spending behavior.
  We are coding using the Google Java programming standard, and our previous programming experience in other classes and on the job experiences.
  The goal is to develop a web application that serves data to front-end pages and also has APIs accessible to read, create, and modify data. Another goal is to deploy this application to Heroku using a CI/CD pipeline, so that code changes can be quickly and efficiently integrated into the project and deployed to the end-users. This code pipeline will also integrate with other automated processes to scan the code for security vulnerabilities, run front-end + back-end tests, and to ensure that best practices are being followed.

- # Software Architecture

  The project uses Apache Maven to handle dependency management, to ensure that the project builds properly, and creates a deployable JAR which will function as the client/server for our application. The application uses the Spring Boot framework to create and initialize the in-memory H2 database, to configure the application based on our settings and to expose REST API endpoints in addition to serving React-based web pages to our users. These REST APIs were designed using Swagger and that YAML document has been provided as a technical artifact.

The application utilizes GitHub for version control and source code management. Through GitHub, a few checks have been implemented in order to ensure code quality prior to a new build.
Once new code has been deployed to the Git repository, the code is packaged into a container using Docker. This container is then deployed on the Heroku platform which allows our users to interact with the application. Two environments have been created for our Heroku app, DEV and PROD. The DEV environment is tied to the GitHub "development" branch and is mainly used for development within an iteration and for testing. PROD is tied to the GitHub "main" branch and is used for release deployments and for the client to interact with our application. Below is a diagram of the client-server architecture which outlines the major components of our application and the technology that it leverages.

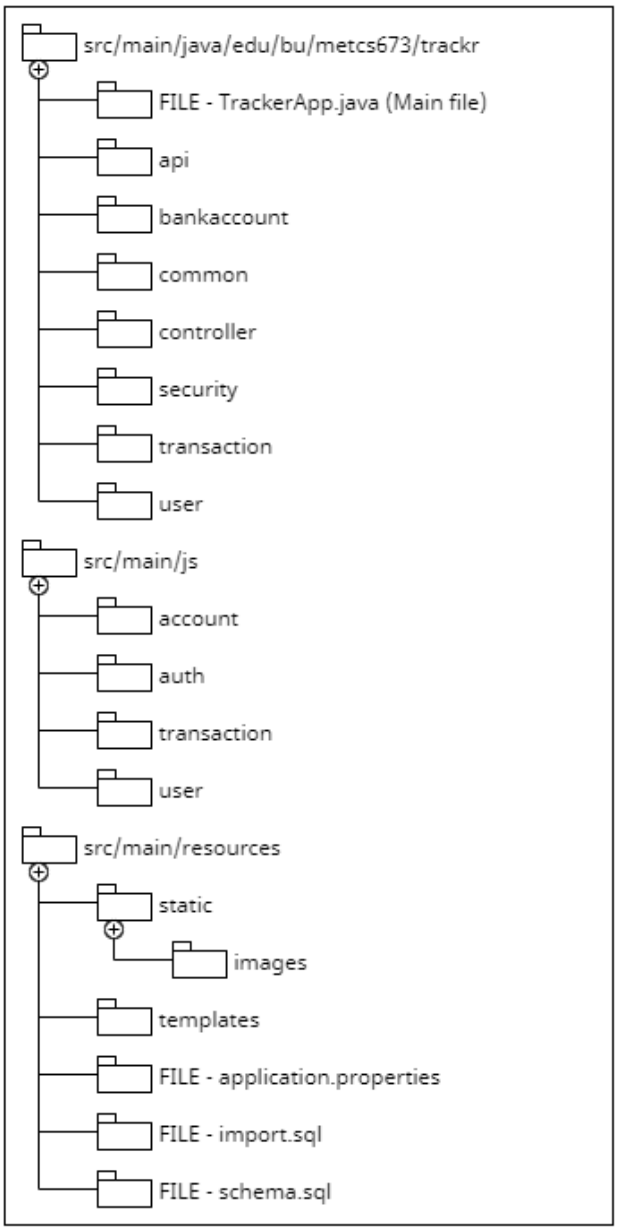


The project is organized using a layered architecture design with a REST transport layer

that registers API calls from the client, a service layer which contains the application business logic, and a data layer that uses JPA to interact with the H2 database.

- ## Class Diagram

  The source code is currently packaged by layer as opposed to by function, however in Iteration 2, the code will be refactored to be packaged by function. Packaging by function will help increase code modularity, cohesion, and make the code easier to navigate as well as make the application adhere to more modern best practices. Each package is a sub-package of the “edu.bu.metcs673.trackr” package, which contains the main class. The project structure can be seen the following diagram:



A brief high-level description of the sub-packages in the “src/main/java” package is provided below.

- Api - Contains a generic response object which is returned by all REST APIs. Contains a generic field for any returned object.
- Bankaccount - contains all classes related to the “BankAccount” resource, which are described below:
    - A domain entity class with Lombok and Java Persistence annotations.
    - A service interface defines methods related to the BankAccount resource, and a service implementation class defines the logic of the related interface methods.
    - A REST Controller class defines the exposed API endpoints for the BankAccount resource.
    - A JPA repository for standard and custom queries to the database for this BankAccount resource.
- Common - Contains files and functionality that applies to multiple resources and source code files. One file contains all static strings that the application uses for success / error messages and others. A custom exception class file is used specifically for our input validations. Another file contains functionality that multiple REST controllers extend.
- Controller - Contains an exception controller which is used for exception handling. Also contains a front-end controller which returns the index file if the application URL is entered into a web browser.
- Security - contains classes relevant to the application’s implementation of JWT Tokens. Includes a filter which is run before each API request to authenticate and authorize the user making the request.
- Transaction - contains all classes related to the “Transaction” resource, which are described below:
    - A domain entity class with Lombok and Java Persistence annotations.
    - A service interface defines methods related to the Transaction resource, and a service implementation class defines the logic of the related interface methods.
    - A REST Controller class defines the exposed API endpoints for the Transaction resource.
    - A JPA repository for standard and custom queries to the database for this Transaction resource.
- User - contains all classes related to the “User” resource, which are described below:
    - A domain entity class with Lombok and Java Persistence annotations.
    - A service interface defines methods related to the User resource, and a service implementation class defines the logic of the related interface methods.
    - A REST Controller class defines the exposed API endpoints for the User

resource.
    - A JPA repository for standard and custom queries to the database for this User resource.

The files located in the “src/main/resources” folder are used during project initialization to create the H2 database and import it with data. The templates folder will contain HTML, CSS, and JavaScript files used by the front-end of this application. These files will use the React framework to make modular and reusable components. The static folder contains static images used in the application’s front-end.

The files in the “src/main/js” folder are classes that define the React front-end of this application. HTML components are defined here to create web pages for the application, additionally custom JavaScript functionality is defined to appropriately capture and translate user action to interact with the Java codebase.

The files in the “src/test/java” folder follow a similar structure to the package structure in “src/main/java” and contain test classes which perform unit tests on the codebase. Mockito is used to mock database calls and to prevent unwanted insertion, modification, or deletion of data.

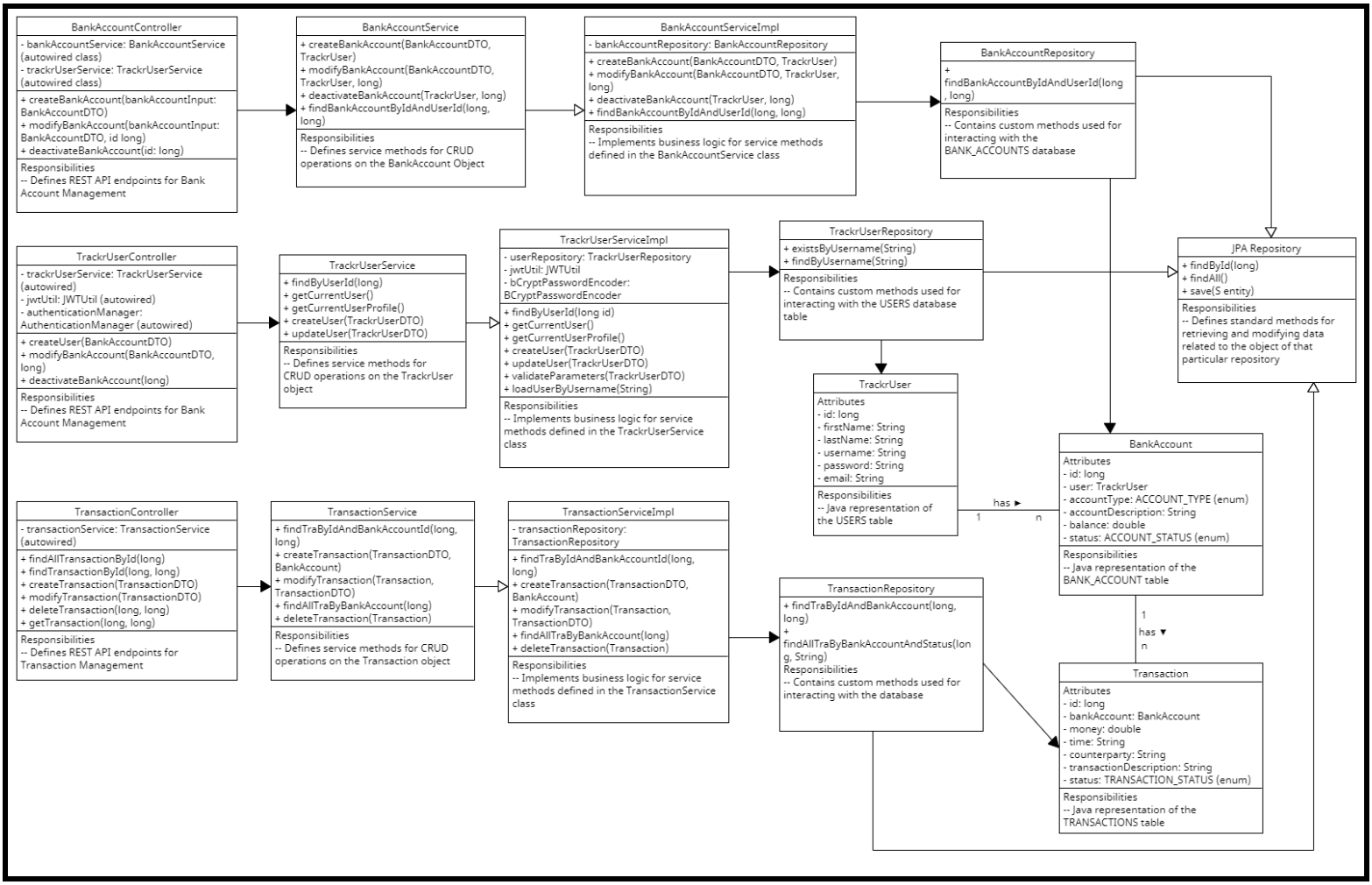
Out application utilizes a closed layered architecture with three main layers:

- Presentation layer - REST API controllers and the React front-end
- Service layer - Contains business logic and processes for validating and manipulating input and output data.
- Data layer - JPA repositories that interact with our database to save or retrieve data from the H2 database.

The presentation layer was designed using OpenAPI standards with Swagger. This document, which is part of the source code, was then used as a reference for the Java implementation. These REST controllers interact and manipulate data through the use of interfaces which are defined and then implemented in a service layer. This layer handles validations, data manipulation, and other business logic for the application. It is abstracted from the presentation layer in order to simplify the purpose of each layer. The service layer implementations then interact with a Data (DAO) layer which takes advantage of JPA functionality in order to communicate with the database, through the use of predefined queries for normal “findBy” and “save” operations, and some custom defined queries for more specific join queries.

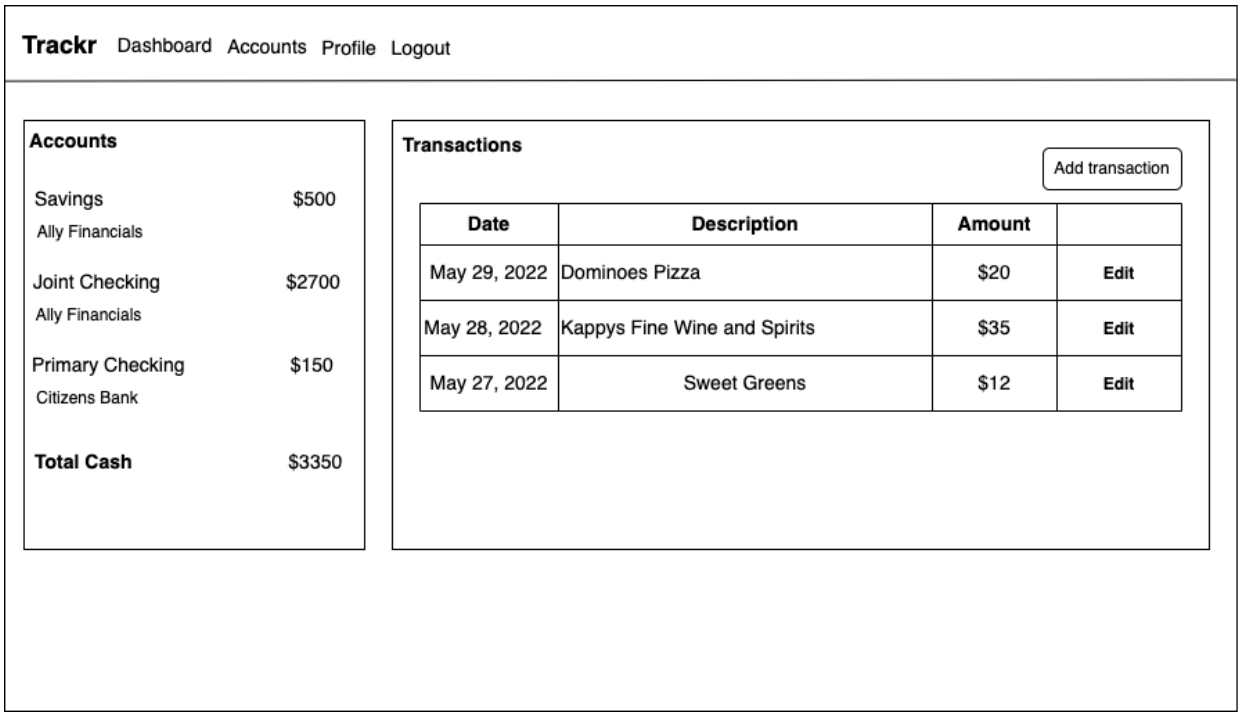
The front-end of this application acts as a component of the presentation layer where a user will take a certain action such as filling out a form or clicking a button to call certain functionality defined in the REST controllers. As of Iteration 2, this front-end has been fully fleshed out and For Iteration 1, the front-end that has been implemented is only for the user registration and login capability. In Iteration 2 and 3, additional functionality will be added so that users can access all API functionality from the React front-end.

The diagram below depicts the layers architecture used for our application and can be viewed as a PNG file using this link.

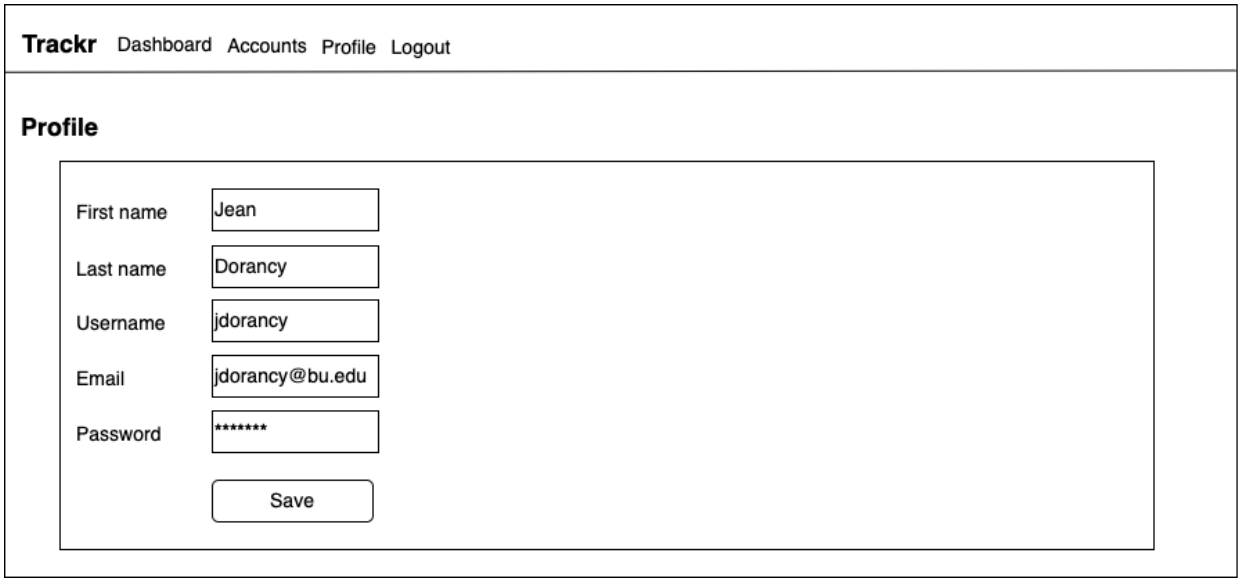


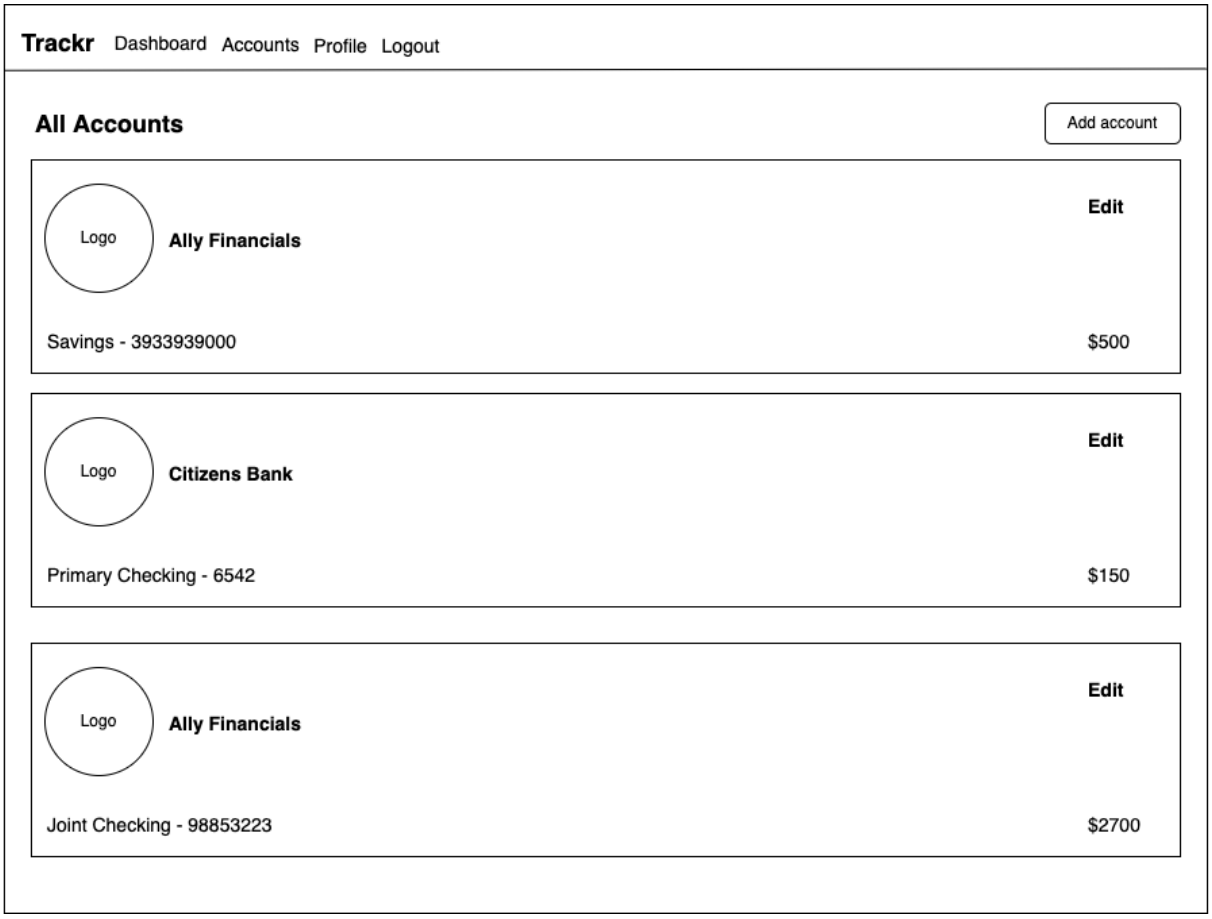
- ## UI Design

  When a user visits the app before they are authenticated they will see two pages. The unauthenticated home page and the login page. These two pages have already been created and will be part of the demo. After login in a user will be redirected to the dashboard page. This page will display bank accounts and balances and a transactions table. The page looks like the following.



There will be two other pages in the application: The user profile page and the all accounts page which will allow users to manage their bank accounts. Please see them below.



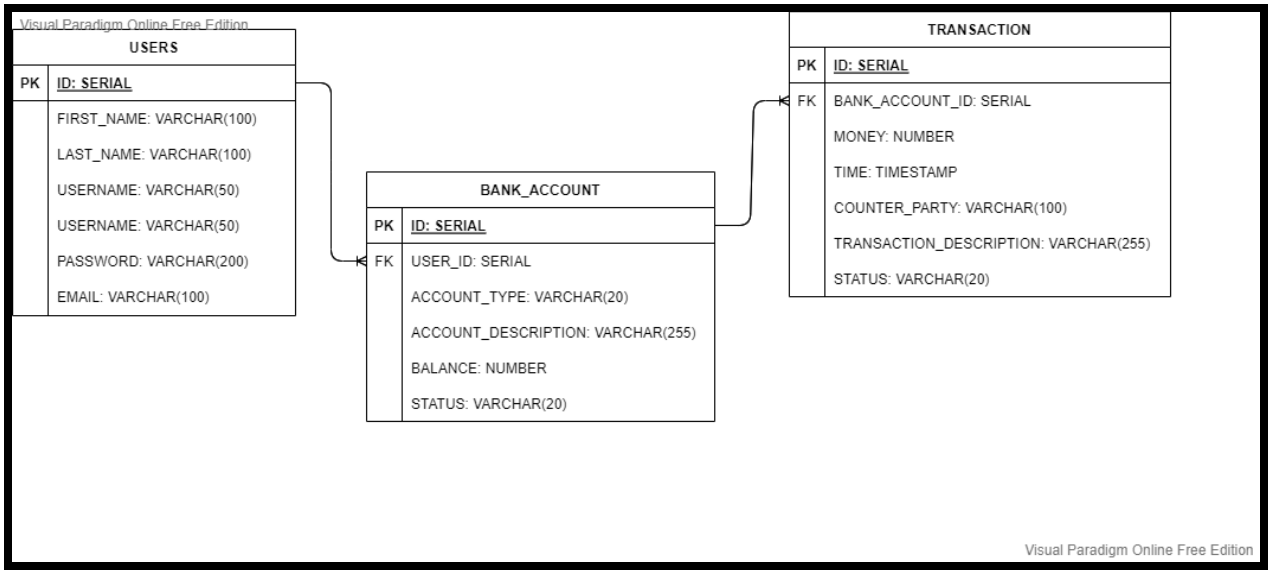


- ## Database Design

  This application utilizes an H2 database, which is a lightweight in-memory database which is very useful for prototypes and small applications since it is easy to configure and integrate with our application's architecture. These configurations are handled by the Spring framework and are accessible through the "application.properties" file in the "src/main/resources" folder. This type of database uses a variation of the SQL dialect and is a relational database. An initial dataset has been created for the application and is created at project runtime with the "schema.sql" and "import.sql" files located in the same directory as the "application.properties" file.

  The Java implementation of this project uses the JPA library to interact with the database in conjunction with Hibernate ORM to map Java objects to their database tables and records.

  An ERD diagram has been provided below to illustrate the database schema and the relationships between tables. It was developed using Visual Paradigm Online.



** The relationships between the tables as of Iteration 1 is Many to One **

- ## Security Design

The application uses the CIA security model with the DAC access-control method, however the user does not have the ability to modify who can access resources associated with themselves. The application's Authentication and Authorization strategy for our REST APIs utilizes bearer tokens, specifically JWT. Upon user registration, a JWT token is returned to the user which will authenticate them for future API calls. Additionally, the password value during registration is encoded using the BCrypt library, specifically using the BCryptPasswordEncoder "encode" method. This method encodes the raw password value with a hash and a salt value.

When this JWT token is provided with an API request, the data in the token is decoded and compared against the data in the USERS table to ensure that the token is valid. Additionally, each token has a specific default "time-to-live" of 15 minutes. After that period of time, the token is invalid and the user will need to hit the "Retrieve Token" API endpoint to get a new valid token. This API endpoint is exposed to the world, but requires the user's specific username and password credentials, set during user registration, in order for a new token to be created.

The JWT token is validated before every API call through the use of a Spring filter, whose logic runs first before the logic of the API endpoint is run. Missing or invalid tokens, will return an "Access Denied" error. If a JWT Token is provided, the USER information is extracted and then used to determine if that record has access to the requested resource. This ability to extract user information from the JWT Token allows

the application to simplify the amount of information needed in the API request. If the provided JWT Token is not associated with the user associated with the requested resource, then the application will return an “Unauthorized Access” error to the user.

Additional security measures are taken when code is pushed to GitHub to ensure that the new code doesn’t break the application as well as not introduce any security risks or vulnerabilities that can be exploited. A few GitHub workflows have been integrated into the code base which effectively build the code and run its front-end (Jest) and back-end (Junit) tests to ensure that the application builds properly and can be deployed safely. CodeQL scans the code base to make sure that there are no security vulnerabilities or risks.

- ## Business Logic

  There is no significant business logic or algorithms taking place in the application, since data is being entered manually and only simple CRUD operations are available to users. Generally, each API request will follow this general algorithm:

  1. The request (API or web) hits one of the REST API endpoints and is intercepted by the JWT filter.
  2. The filter checks for the JWT token in order to authenticate the user.
  3. If the user is authenticated, then the data in the request itself is validated
     a. Null checks on required fields
     b. Type checks for numeric fields
     c. Value checks for fields that are enumerations
     d. API-specific validations
  4. For GET requests, once data is returned, the user data from the JWT token is compared against the user data associated with the record. If there is a match, then data is returned. If there is a mismatch, then an “Access Denied” error is returned.
  5. For POST requests, once the data is written to the database, a response body is returned to the user indicating their success.
  6. For PUT and DELETE requests, an id value provided as a path variable is used to retrieve the data that will be changed. If this data is associated with the user making the API call, then that record is modified and a response is returned to the user indicating their success.

- ## Design Patterns

  The Trackr application uses an MVC design pattern and layered architecture with the Spring framework in order to initialize the application, expose the REST API endpoints and interact with the database. Under the hood, the application takes advantage of many of Springs annotations to autowire the service layer which creates singleton instances of

important classes such as the “TrackrUserService”, “TrackrUserServiceImpl”, and the “TrackrRepository”.

The application also uses a Factory creation design pattern to handle REST API response objects that are returned to the user. Since the only outcome of such a request is either success or failure, the “GenericApiResponse” class has two methods which return these responses, but can be customized using the “message” and “object” method parameters. This allows for the application to return specific messages to the user such as a variety of error messages or conversely, a newly modified object for the user to review.

React Container Pattern for the Frontend
This is simply about separating React components that are responsible for "presentation" from the logic for fetching data, processing and working with the backend. The pattern is implemented as follows.

- **Single Responsibility Principle**: Build components that are focused on one thing. For example, the **LoginForm** is just that and nothing else. Keep in mind this also enables components re-usability.
- **Service Class for API**: The service class simply abstract away how to work with the backend API.
- **Container Component**: This renders the component which does "presentation" and also calls methods from the service class. For example **HomeContainer** which renders **Home** and uses **TrackrUserService**.
- **Presentation**: Simply showing the data and inputs to the user as desired.

References

- Mint UI is used as inspiration
  https://mint.intuit.com/
- Understanding the Container Component Pattern with React Hooks
  https://blog.openreplay.com/understanding-the-container-component-pattern-with-react-hooks

Glossary

Acronyms and abbreviations:

- **IDE** - Integrated Development Environment
- **YAML** - YAML ain’t Markup Language
- **MVC** - Model, View, Controller
- **DB** - Database
- **API** - Application Programming Interface
- **REST** - Representational State Transfer

- **CRUD** - Create, Read, Update, Delete
- **CI** - Continuous Integration
- **CD** - Continuous Deployment (Delivery)
- **JPA** - Java Persistence API
- **JWT** - Java Web Token
- **ORM** - Object Relational Mapping
- **SQL** - Structured Query Language
- **CIA** - Confidentiality, Integrity, Availability
- **DAC** - Discretionary Access Control