University of Houston-Victoria

Department of Computer and Information Science

Group Project

Expense Tracker

Software Design Specification

Prepared for Partial Fulfillment of the Course

COSC 6342 Software Engineering Project Management

Spring 2024

By

Aradhana Sharma (2250955)

Kapil Suryawanshi (2039409)

Shayan Khan (2244554)

Suraj Odedra (2329291)

Advisor: Dr. Amjad Nusayr

Director of UG Computer Science & Computer Information Systems  
Associate Professor of Computer Science

University of Houston Victoria

Table of Contents

[**I.** **Abstract** 4](#_Toc162723836)

[**II.** **High Level Architecture** 5](#_Toc162723837)

[**III.** **Low Level Architecture** 6](#_Toc162723838)

[**a.** **Front End** 6](#_Toc162723839)

[**b.** **Back End** 8](#_Toc162723840)

[**IV.** **Detailed Design** 10](#_Toc162723841)

[**a.** **Database** 10](#_Toc162723842)

[**b.** **Front End Mockups** 11](#_Toc162723843)

[**c.** **Front End** 20](#_Toc162723844)

[**d.** **Back End** 22](#_Toc162723845)

[**e.** **Activity Diagrams** 23](#_Toc162723846)

[**Use case: Create transaction category** 24](#_Toc162723847)

[**Use case: Edit my transaction category** 28](#_Toc162723848)

[**Use case: View my transaction categories** 31](#_Toc162723849)

[**Use case: Delete my transaction category** 34](#_Toc162723850)

[**V.** **Contributions** 35](#_Toc162723851)

[**VI.** **Glossary** 35](#_Toc162723852)

[**VII.** **References** 36](#_Toc162723853)

Table of figures

[Figure 1 High Level Architecture 5](#_Toc162723854)

[Figure 2 Low Level Architecture - Front End 6](#_Toc162723855)

[Figure 3 Low Level Architecture - Back End 8](#_Toc162723856)

[Figure 4 ER diagram 10](#_Toc162723857)

[Figure 5 Login screen 12](#_Toc162723858)

[Figure 6 Registration screen 12](#_Toc162723859)

[Figure 7 Dashboard for Transactions screen 13](#_Toc162723860)

[Figure 8 Dashboard for Group Transactions screen 14](#_Toc162723861)

[Figure 9 Categories listing screen 15](#_Toc162723862)

[Figure 10 Category create/edit screen 15](#_Toc162723863)

[Figure 11 Budgets listing screen 16](#_Toc162723864)

[Figure 12 Budget create/edit screen 16](#_Toc162723865)

[Figure 13 Transactions listing screen 17](#_Toc162723866)

[Figure 14 Transactions create/edit screen 17](#_Toc162723867)

[Figure 15 Groups listing screen 18](#_Toc162723868)

[Figure 16 Group create/edit screen 18](#_Toc162723869)

[Figure 17 Group Transactions listing screen 19](#_Toc162723870)

[Figure 18 Group Transaction create/edit screen 19](#_Toc162723871)

[Figure 19 Detailed design - Front End 20](#_Toc162723872)

[Figure 20 Detailed design - Back End 22](#_Toc162723873)

[Figure 21 Category management use cases 23](#_Toc162723874)

[Figure 22 Activity diagram - Create transaction category 24](#_Toc162723875)

[Figure 23 Signals/Event and Receptions/Event handler symbols 25](#_Toc162723876)

[Figure 24 Activity Diagram - Create transaction category - Part 1 26](#_Toc162723877)

[Figure 25 Activity Diagram - Create transaction category - Part 2 27](#_Toc162723878)

[Figure 26 Activity Diagram - Edit transaction category 28](#_Toc162723879)

[Figure 27 Activity Diagram - Edit transaction category - Part 1 29](#_Toc162723880)

[Figure 28 Activity Diagram - Edit transaction category - Part 2 30](#_Toc162723881)

[Figure 29 Activity Diagram - View my transaction categories 31](#_Toc162723882)

[Figure 30 Activity Diagram - View my transaction categories - Part 1 32](#_Toc162723883)

[Figure 31 Activity Diagram - View my transaction categories - Part 2 33](#_Toc162723884)

[Figure 32 Activity Diagram - Delete my transaction category 34](#_Toc162723885)

Table of tables

[Table 1 Glossary 36](#_Toc162723886)

# **Abstract**

The Expense Tracker web application is a comprehensive personal finance management solution designed to simplify the process of tracking expenses, budgeting, and financial planning for individuals and groups. Built with ReactJS and ExpressJS, this innovative platform offers a robust set of features including real-time expense tracking, customizable budgeting tools, and interactive financial reports. The application's unique group expense management feature allows for seamless splitting and tracking of shared costs, making it ideal for families, roommates, and other collaborative financial arrangements. With a strong focus on user-centric design and personalized experiences, the Expense Tracker is committed to empowering users with valuable insights and fostering greater financial well-being.

The project's design specification encompasses high and low-level architecture details, database design, front-end and back-end design specifications, use case activity diagrams, and team contributions, ensuring a thorough and well-structured approach to development.

# **High Level Architecture**

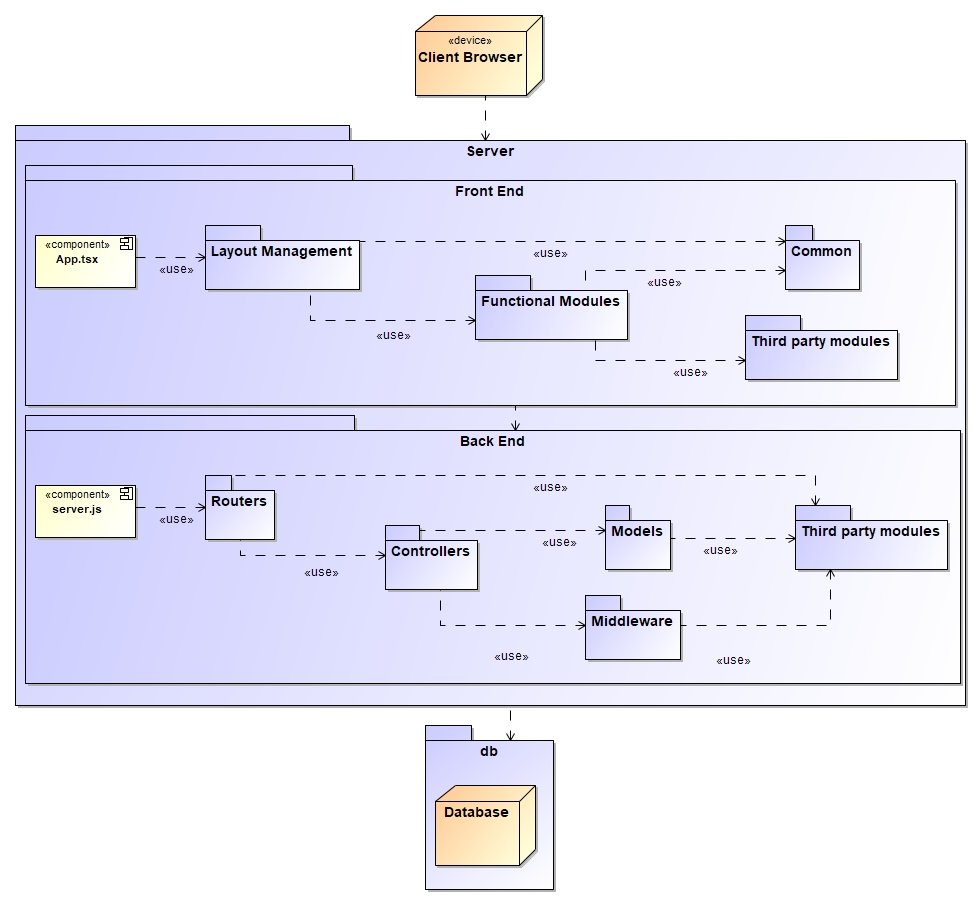


Figure High Level Architecture

Figure 1 shows the high level architecture for Expense Tracker web application, which is a typical client-server web application. The architecture diagram represents a web application with a client-side front-end developed using ReactJS and a server-side backend developed using ExpressJS. The front-end is responsible for rendering the user interface and handling user interactions, while the backend handles server-side logic, data processing, and communication with the database.

The central component is the "Client Browser," where users interact with the application through a web browser. The front-end ReactJS code is responsible for rendering the user interface components, managing the application state, and handling user events.

The "Server" component represents the backend ExpressJS application, which handles server-side operations and communicates with the front-end and the database. It consists of various modules, such as "Routers," "Controllers," "Models," and "Middleware," following a typical ExpressJS-NodeJS application architecture pattern.

The "Layout Management" and "Functional Modules" components are part of the front-end codebase, responsible for managing the layout and implementing the application's core functionality, respectively. The "Third Party Modules" represent external libraries or services integrated into both the front-end and backend components.

The "Database" component represents the data storage system, which can be accessed and manipulated by the backend ExpressJS application. The lines connecting the components indicate the dependency and control between them, enabling seamless communication and data exchange within the application.

# **Low Level Architecture**

## **Front End**

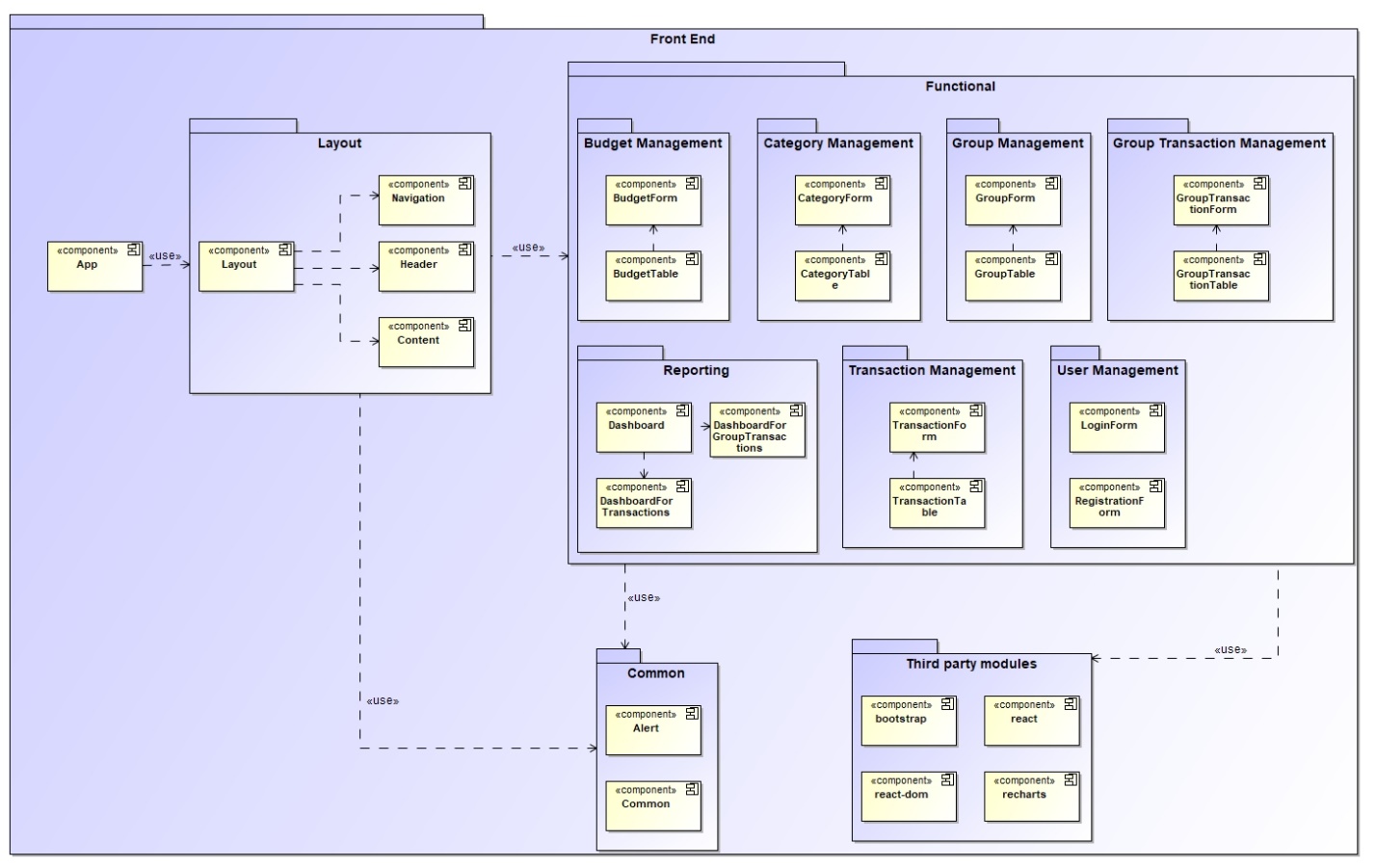


Figure Low Level Architecture - Front End

Figure 2 shows the detailed architecture for the front end of the Expense Tracker application. The front end will be developed in ReactJS UI framework. This framework is component based framework, therefore different components will be designed which represent different parts/sections of the front end.

The front end consists of prominent modules namely Layout, Functional and Common. The Layout module contains sub components which establishes the application layout. The Functional module contains UI components responsible for interacting with the core functionality of the application. The Common module contains common functions, enumerations and UI components for supporting the Functional module.

The diagram above, depicts all these components and their organizations. Each of these components are described below.

* **Layout Management**:
  + This component handles the overall layout and structure of the user interface.
  + It includes sub-components like Navigation, Header, and Content, which are responsible for rendering different sections of the UI.
* **Functional Modules**:
  + These are the core components that provide the main functionality of the application.
  + **Budget Management**: Handles budget-related operations like creating, updating, and displaying budgets. It includes sub-components like BudgetForm and BudgetTable.
  + **Category Management**: Manages categories for expenses or income. It has sub-components like CategoryForm and CategoryTable.
  + **Group Management**: Handles the creation and management of user groups. It includes sub-components like GroupForm and GroupTable.
  + **Group Transaction Management**: Manages transactions within user groups. It has sub-components like GroupTransactionForm and GroupTransactionTable.
  + **Reporting**: Provides reporting and visualizing functionality. It includes sub-components like Dashboard for displaying reports and visualizations.
  + **Transaction Management**: Handles individual user transactions. It has sub-components like TransactionForm and TransactionTable.
  + **User Management**: Manages user accounts, authentication, and authorization. It includes sub-components like LoginForm and RegistrationForm.
* **Common**:
  + This component contains shared functionality used across multiple modules.
  + It includes sub-components like Alert for displaying notifications and Common for shared utilities or services.
* **Third-party Modules**:
  + These are external libraries or modules integrated into the application.
  + It includes sub-components like bootstrap for UI styling, react-dom for rendering React components on the DOM, and recharts for creating charts and visualizations.

The architecture follows a modular approach, where each functional module is broken down into smaller, reusable components. The Layout Management component provides the overall structure, while the Functional Modules handle the core application logic. The Common and Third-party Modules provide shared functionality and external dependencies, respectively. This architecture promotes code reusability, maintainability, and scalability, which are essential for large-scale React applications.

## **Back End**

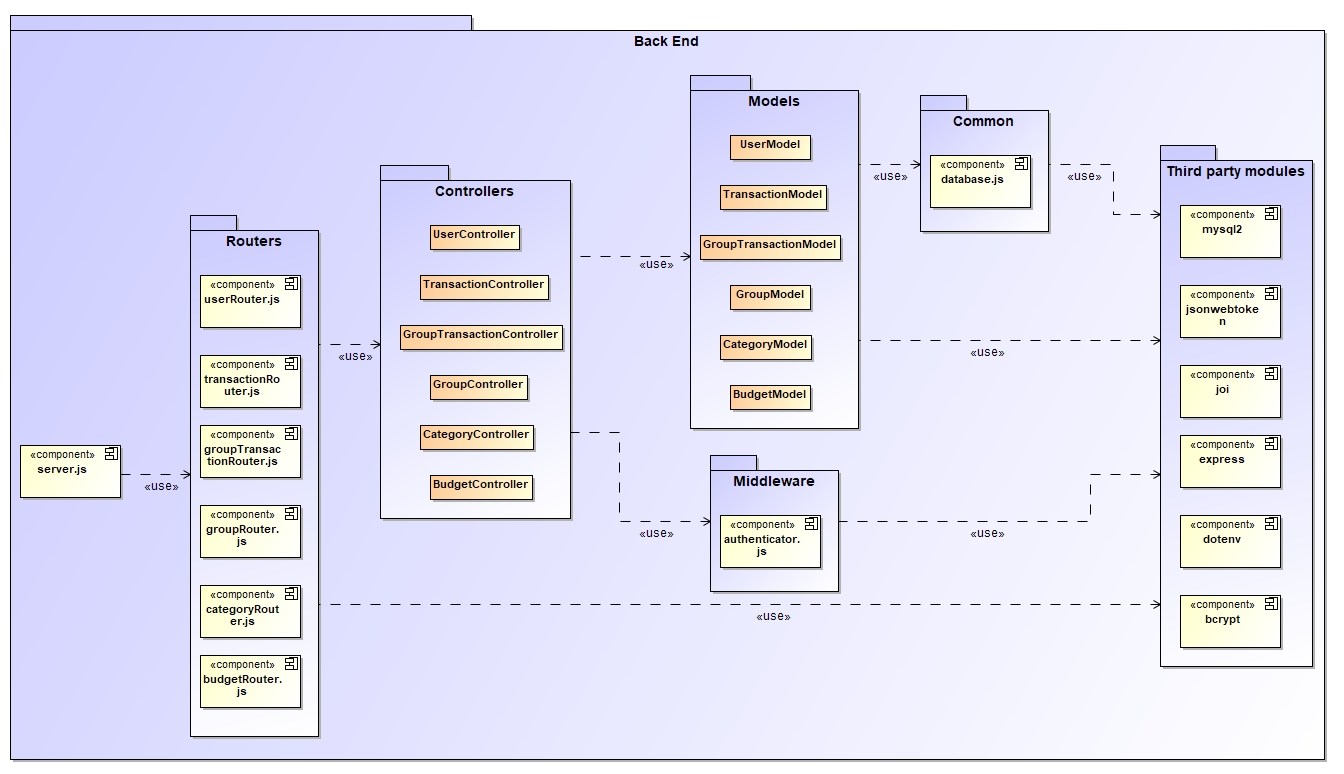


Figure Low Level Architecture - Back End

Figure 3 shows the low level architecture of Expense Tracker application. Following are the details about the elements depicted.

The back end will be a layered architecture with prominent layers of Router, Controller and Model. The router layer will be responsible for routing the web service end points to respective controller functions. The controller layer will be responsible for dealing with HTTP requests and response to and from client. The model layer will be responsible for communicating with database to fetch and update data. The following is the information about the subcomponents of these layers.

* **Routers**
  + **userRouter.js**: This router handles user-related routes and endpoints, such as user registration, authentication, and profile management.
  + **transactionRouter.js**: This router is responsible for handling routes and endpoints related to transactions, such as creating, updating, and retrieving transaction data.
  + **groupTransactionRouter.js**: This router manages routes and endpoints for group transactions, which could involve multiple users or entities.
  + **categoryRouter.js**: This router handles routes and endpoints related to categories, which could be used for organizing transactions.
  + **budgetRouter.js**: This router is responsible for routes and endpoints related to budgets, against the user categories.
* **Controllers**
  + **UserController**: This controller handles the HTTP request and response related to user management, such as user authentication, registration, and profile updates.
  + **TransactionController**: This controller manages the HTTP request and response for transactions, including creating, updating, and retrieving transaction data.
  + **GroupTransactionController**: This controller handles the HTTP request and response for group transactions, which may involve multiple users or entities.
  + **GroupController**: This controller handles the HTTP request and response related to user groups within the application.
  + **CategoryController**: This controller handles the HTTP request and response for categories, such as creating, updating, and retrieving category data.
  + **BudgetController**: This controller handles the HTTP request and response related to user budgets, against user categories for transactions.
* **Models**
  + **UserModel**: This model represents the business logic, operations, data structure and schema for user data, including user profiles, authentication information, and other user-related details.
  + **TransactionModel**: This model represents the business logic, operations, data structure and schema for transaction data, such as transaction details, amounts, dates, and associated entities.
  + **GroupTransactionModel**: This model represents the business logic, operations, data structure and schema for group transactions, which may involve multiple users or entities.
  + **GroupModel**: This model represents the business logic, operations, data structure and schema for user groups within the application.
  + **CategoryModel**: This model represents the business logic, operations, data structure and schema for categories, which could be used for organizing user transactions.
  + **BudgetModel**: This model represents the business logic, operations, data structure and schema for user budget data, against user category for transactions.
* **Common**
  + **database.js**: This components creates a connection pool to the database, allowing the application to interact with the database.
* **Middleware**
  + **authenticate.js**: This middleware is responsible for authentication and authorization, ensuring that only authenticated users can access certain routes or perform specific actions within the application.
* **Third-party modules**
  + The diagram shows several third-party modules or libraries that the application is using, such as `mysql2`, `jsonwebtoken`, `bcryptjs`, `express`, `joi` and `dotenv`. These modules provide additional functionality or utilities for the application, such as database integration, authentication, validation, and configuration.

# **Detailed Design**

## **Database**

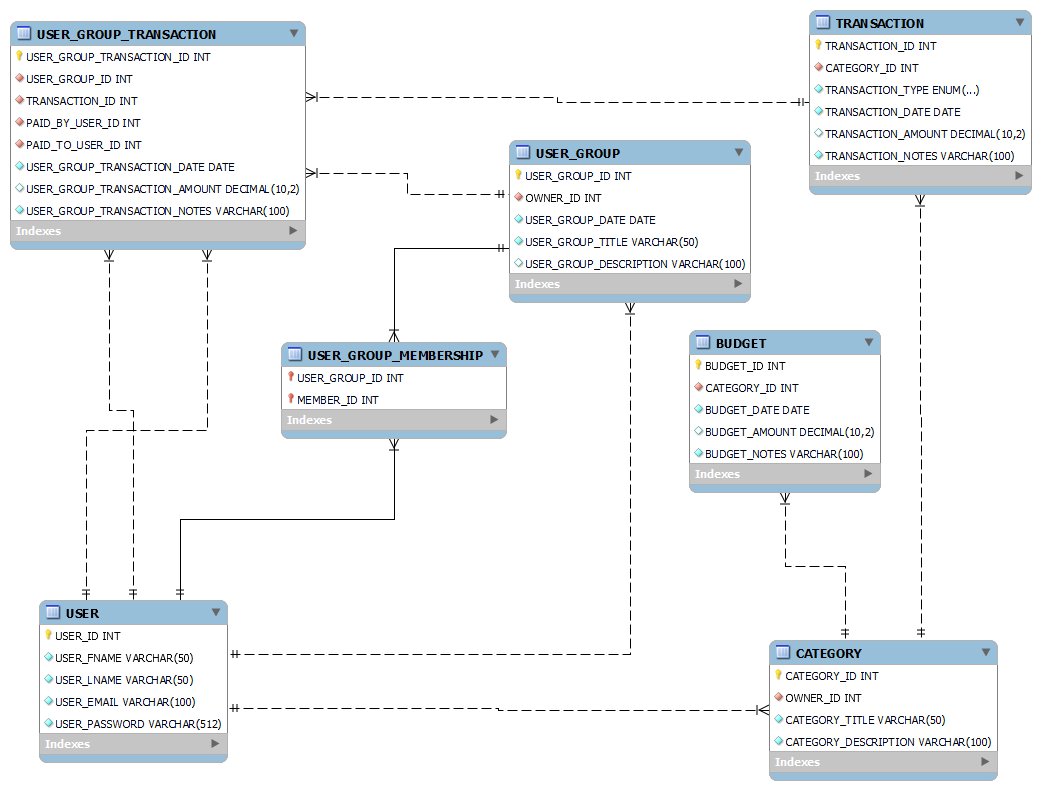


Figure ER diagram

Figure 4 shows the Entity Relationship Diagram (ERD) for Expense Tracker application. The data model depicts a primary key of a table with a key symbol, while foreign keys are represented by red diamond symbols.

This data model is designed to adhere to the following business rules of this application.

* 1. A user can own zero or more categories.
  2. A category belongs to one user only.
  3. A category can have zero or more budget configured.
  4. A budget is only for one category.
  5. A category can categorize zero or more transactions.
  6. A transaction can be categorized by one category.
  7. A transaction can be split into zero or more group transactions.
  8. A group transaction can trace back to only one main transaction.
  9. A user group can have zero or more group transactions.
  10. A group transaction belongs to only one user group.
  11. A user can own zero or more user groups.
  12. A user group belongs to only one user.
  13. A user group can have one or more users as members.
  14. A user can be member of zero or more user groups.
  15. A user group transaction can have one user who pays and one user who receives the money.
  16. A user can be a payer in zero or more user group transactions.
  17. A user can be a received in zero or more user group transactions.

Along with these business rules, the database schema will be configured with application specific uniqueness and not null constraints as well.

## **Front End Mockups**

Following are the screen mockups for the user interface design. At high level the Expense Tracker application would be used in following steps.

* + - 1. User registers an account with the application, if not already done.
      2. User logins to the application with his/her credentials.
      3. User will create Categories in the application. The user can list, edit and delete the Categories, as and when the required business rules are satisfied.
      4. User will create Budgets in the application against the Categories created earlier. The user can list, edit and delete the Budgets, as and when the required business rules are satisfied.
      5. User will create Transactions in the application against the Categories created earlier. The user can list, edit and delete the Transactions, as and when the required business rules are satisfied.
      6. User will create Groups in the application, with selected members from existing Users from the application. The user can list, edit and delete the Groups, as and when the required business rules are satisfied.
      7. User will create Group Transactions in the application against the Categories and Groups created earlier. The user can list the Group Transactions and visualize the settlement summary.
      8. User will be able to view the visual reports from the dashboard of the applications when needed.

With this as the overall user interaction with the Expense Tracker application, following mockup screens will provide overall user experience.

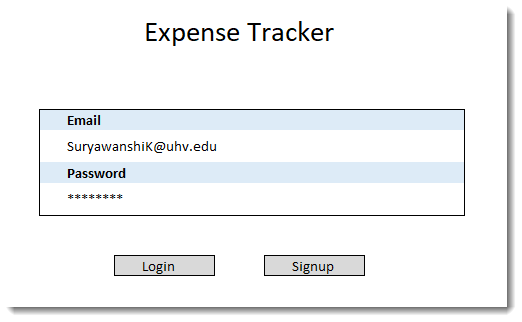


Figure Login screen

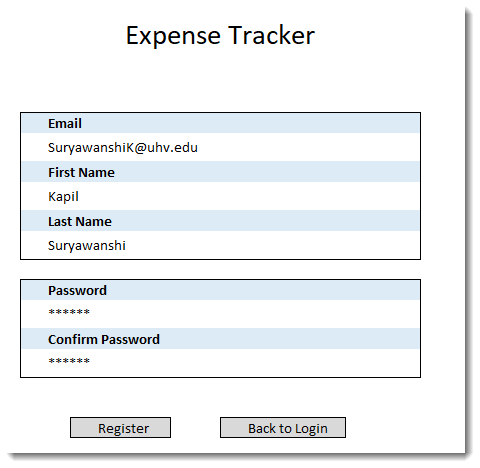


Figure Registration screen



Figure Dashboard for Transactions screen

This dashboard is aimed at showing following 4 charts for user transactions to understand and get insights of his/her cash flow. This dashboard will show a data for a month.

* Income
  + Pie chart would show the actual income per income Category.
  + Bar chart would show the actual income per income Category and Budget.
* Expense
  + Pie chart would show the actual expense per expense Category.
  + Bar chart would show the actual expense per expense Category and Budget.



Figure Dashboard for Group Transactions screen

This dashboard is aimed at showing following 2 charts for user group transactions to understand and get insights of his/her cash flow. These are unsettled group transactions. This dashboard will consider overall data.

* Money owed to me
  + Bar chart to show the money owed to me by each person.
* Money I need to pay
  + Bar chart to show the money I need to pay to each person.

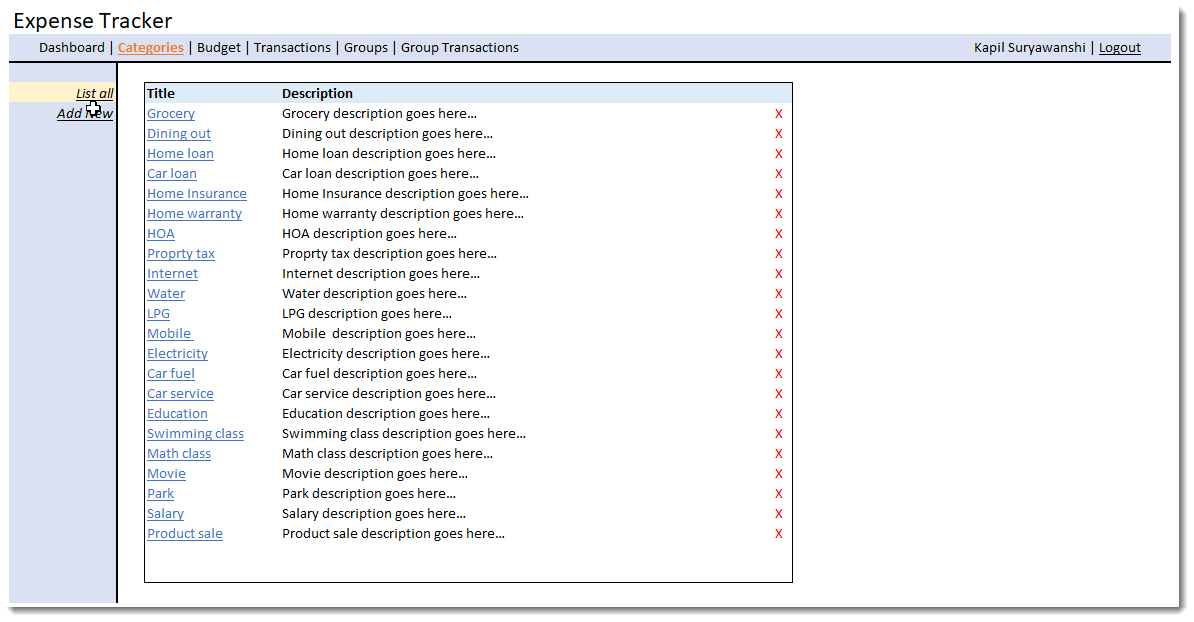


Figure Categories listing screen

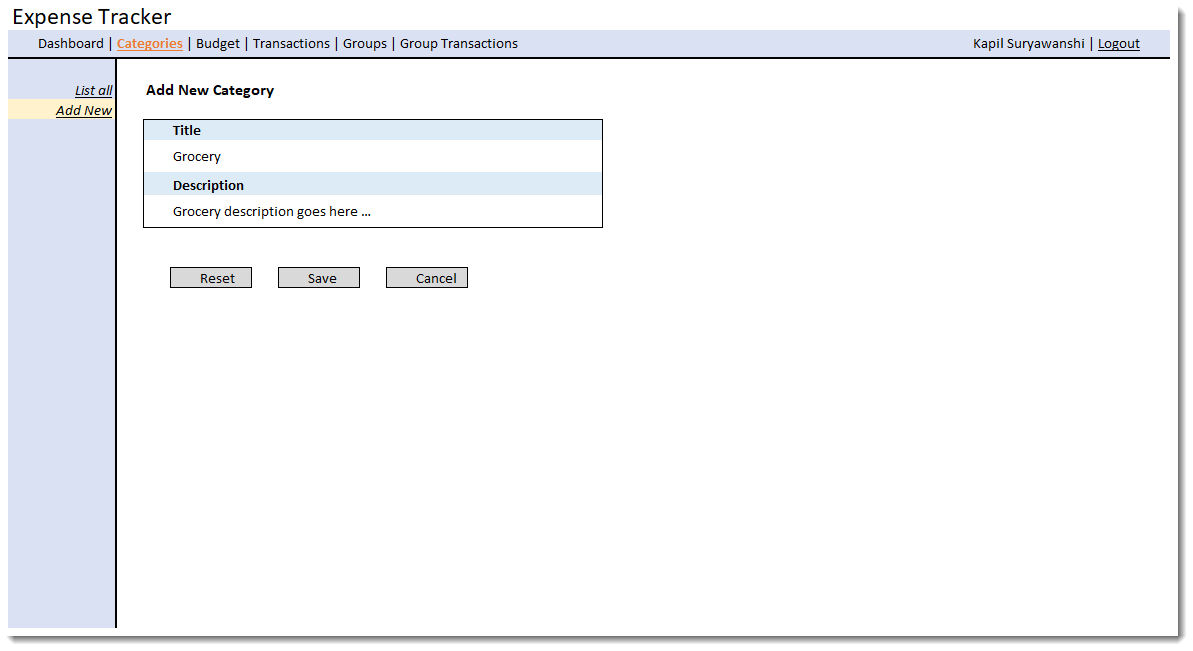


Figure Category create/edit screen

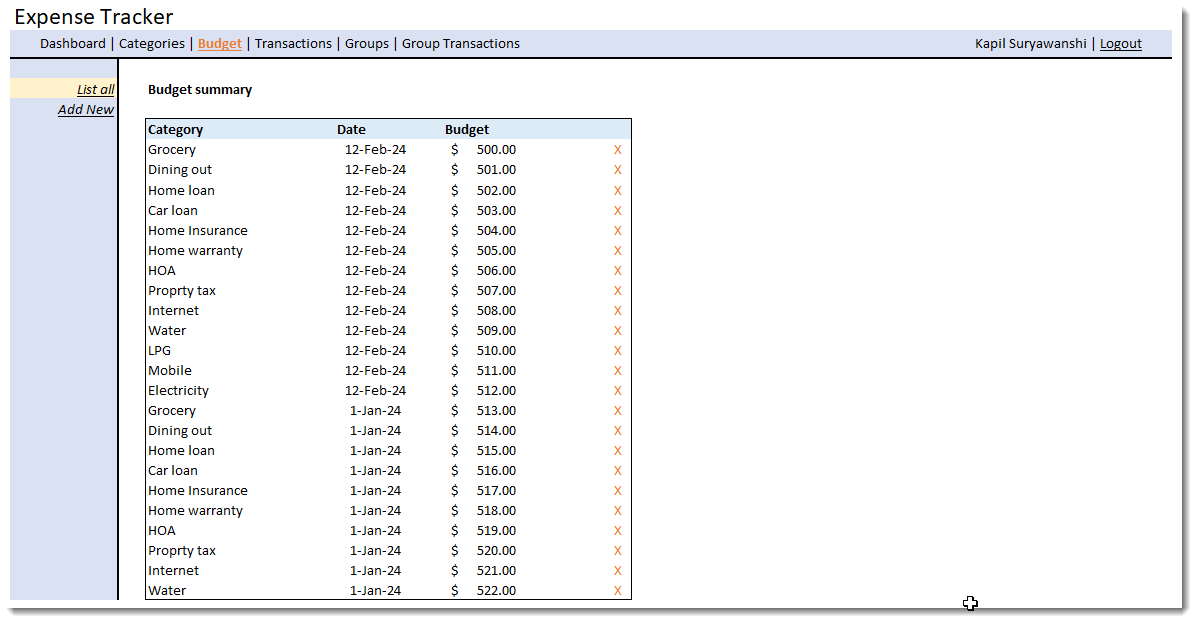


Figure Budgets listing screen

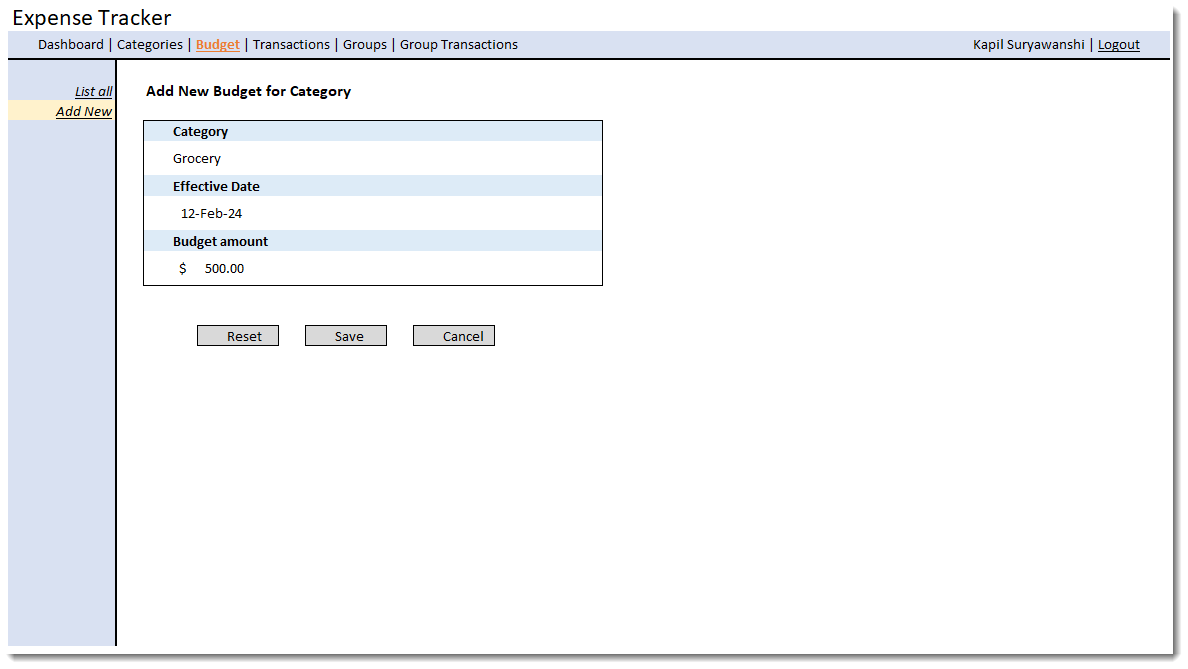


Figure Budget create/edit screen

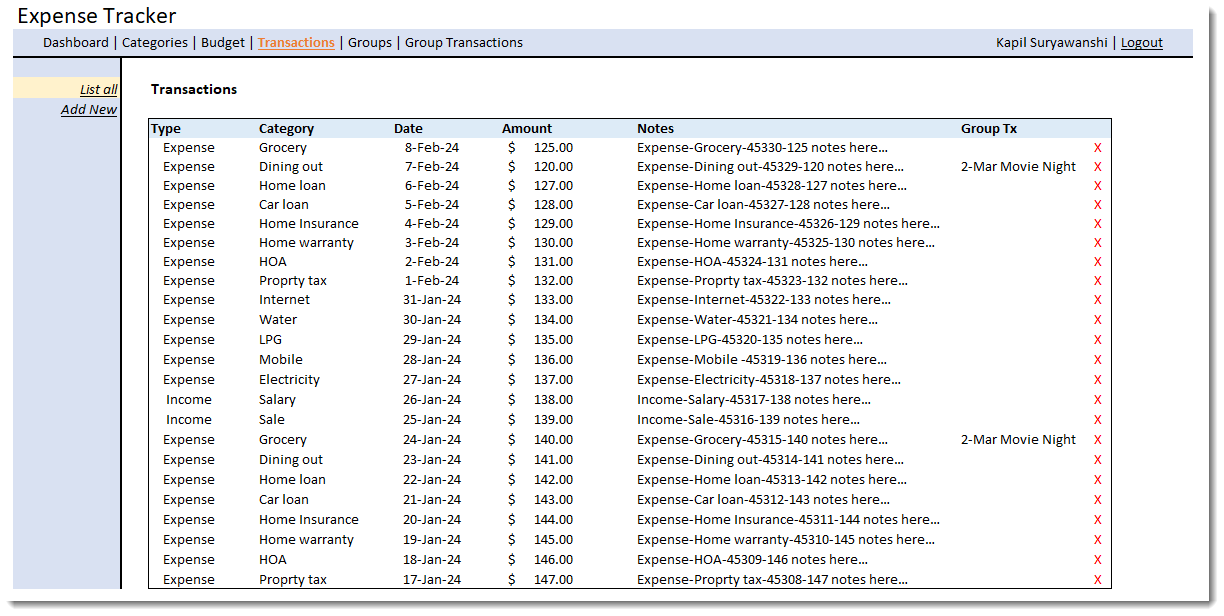


Figure Transactions listing screen

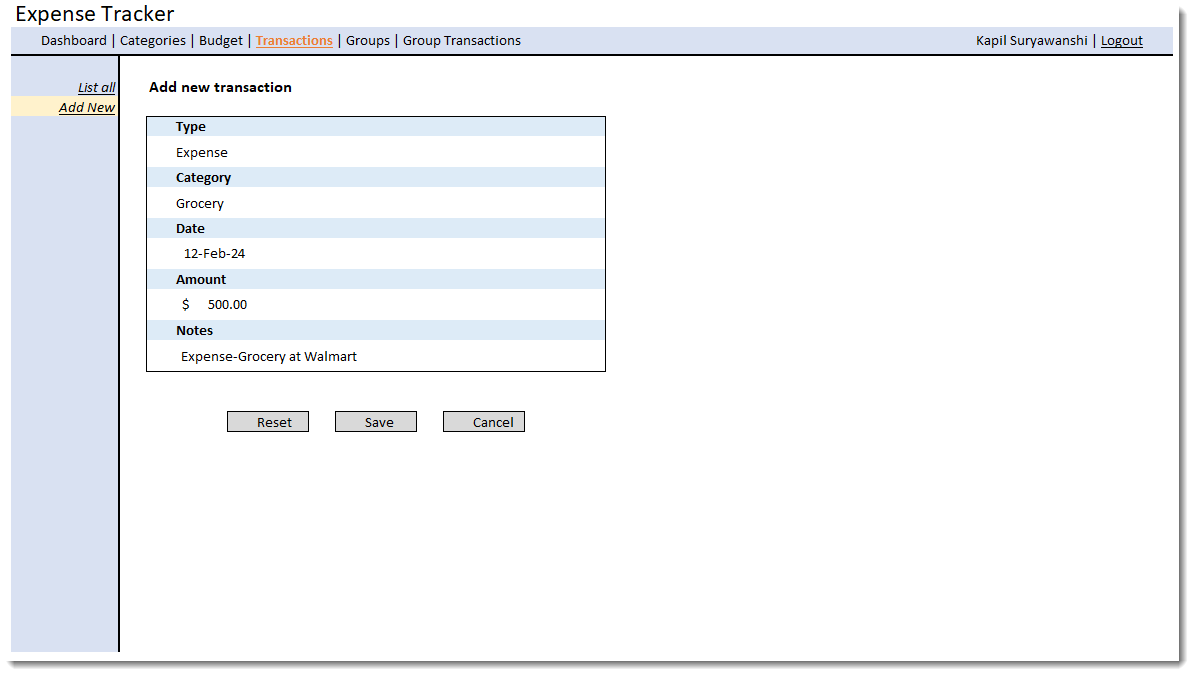


Figure Transactions create/edit screen

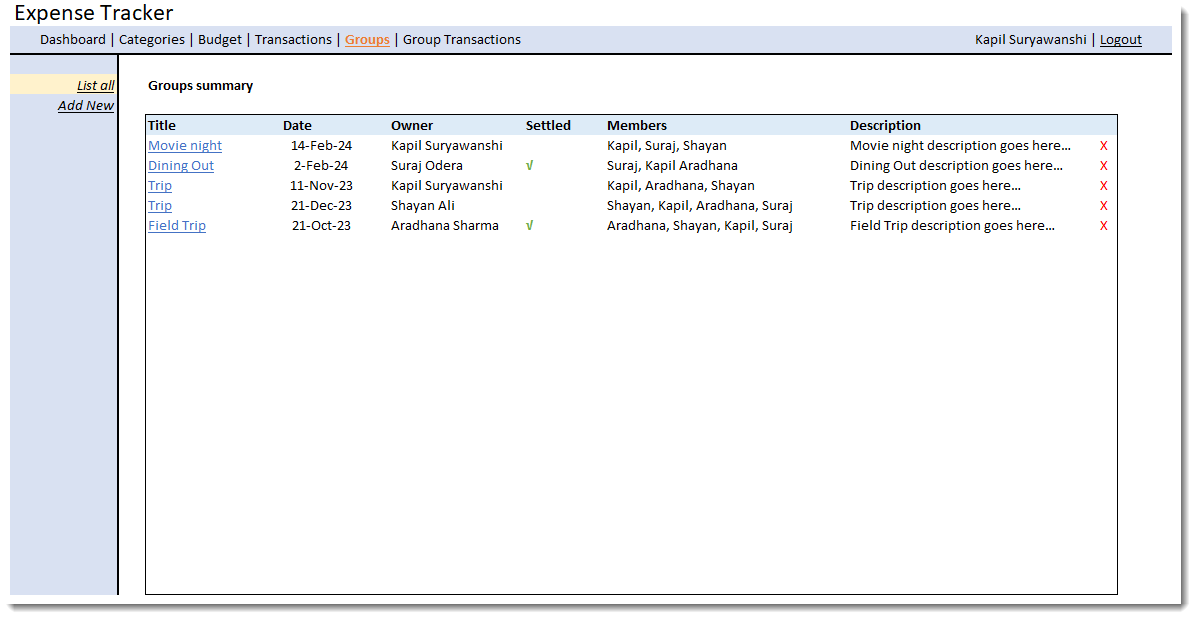


Figure Groups listing screen

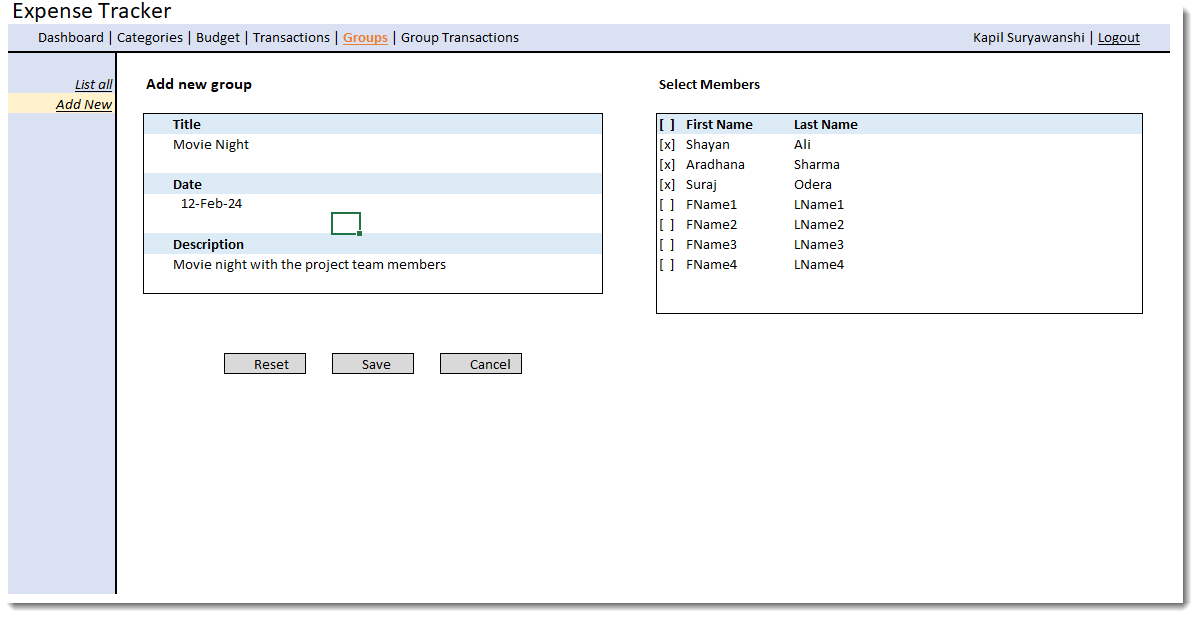


Figure Group create/edit screen

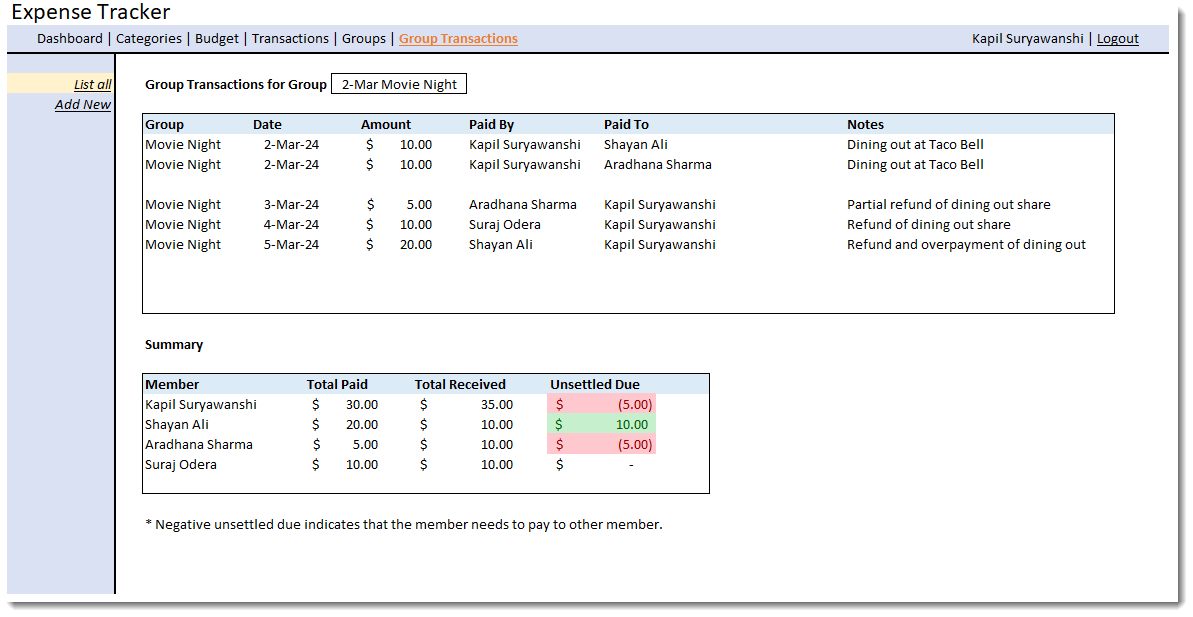


Figure Group Transactions listing screen

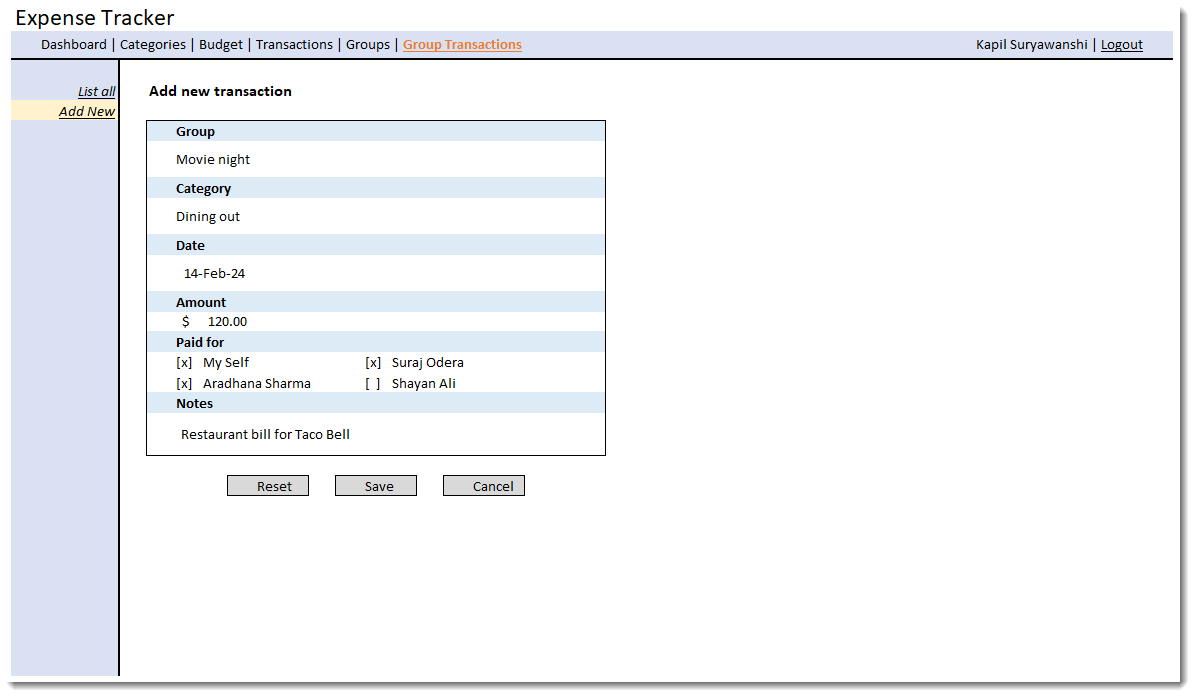


Figure Group Transaction create/edit screen

## **Front End**

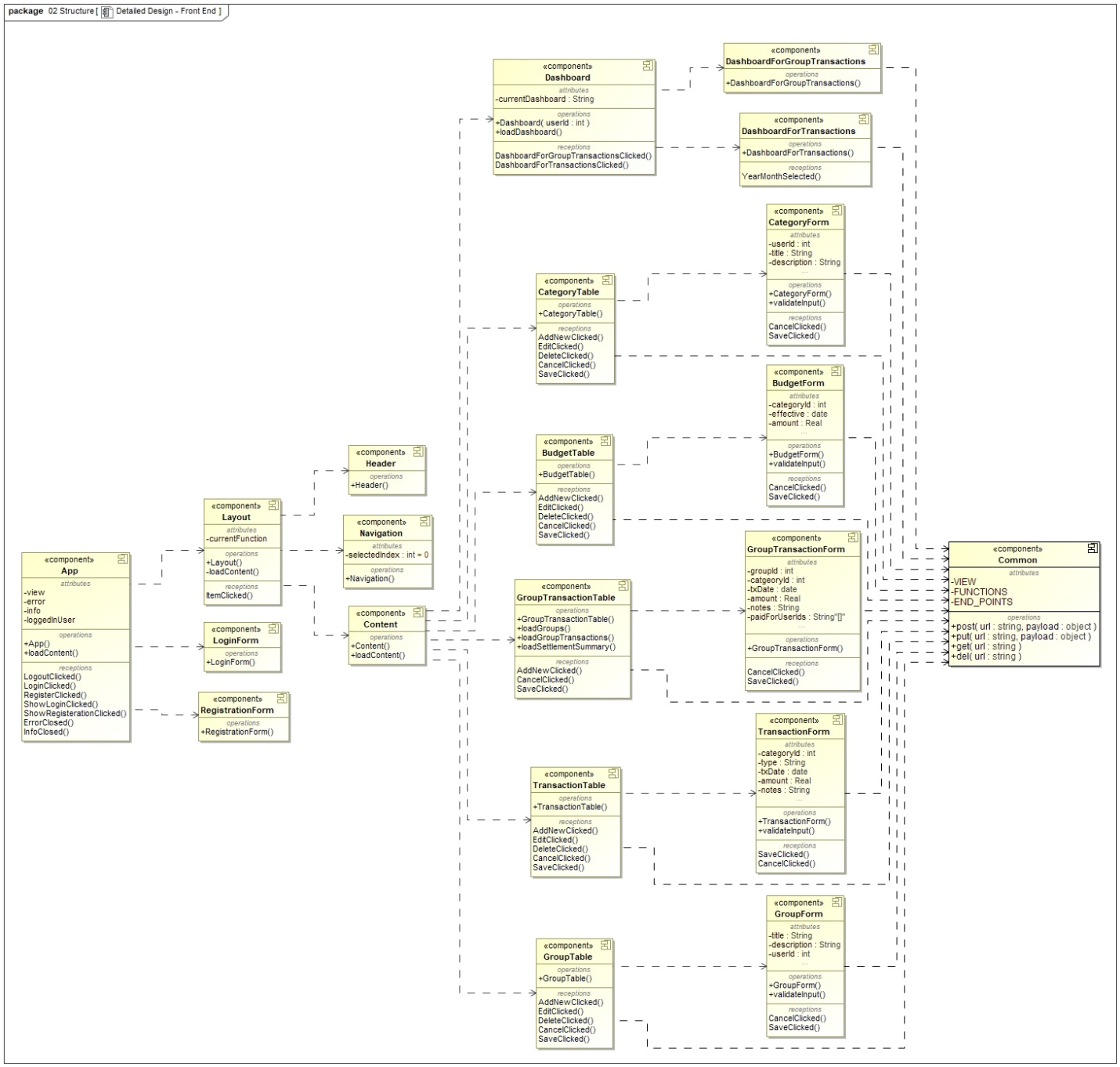


Figure Detailed design - Front End

Figure 19 illustrates an UML Component diagram, detailing the design of components involved in front-end development. Each entity in this design represents a UML Component, with dotted lines with arrows depicting the development dependencies between components. The component at one end of the relationship is a client component, while the component at the arrow end is a supplier component.

This depiction of components aligns naturally with the chosen technology, ReactJS, utilized for front-end development. In ReactJS, the UI is constructed in terms of reusable components, which handle event management, state, and UI rendering. A UI component achieves its rendering by aggregating other UI components if necessary.

(As ReactJS components are not Class constructs but rather functions in JavaScript, they cannot be represented as UML Classes.)

The attribute section of each component lists the state variables, while the operation section represents functions that accomplish specific objectives. The reception section denotes event handlers owned by the component.

The App component serves as the entry point of the front end, making the ReactJS front end a single page JavaScript application. Upon application start, three possibilities exist: displaying the Registration screen, Login screen, or the home page.

The home page is structured by Layout component, which include Navigation, Header, and Content components. The Content component loads functional components based on user interaction with the front end.

Functional components are designed as XXXTable and XXXForm components (e.g., CategoryTable, CategoryForm). Initially, XXXTable components list data objects in tabular form, with each table row representing an object in the application. These rows offer actions to edit and delete objects based on the application's business rules. Additionally, XXXTable components feature an "Add New" action, invoking XXXForm components to accept user data and create corresponding objects in the application database.

In addition to these functional components, Dashboard components exist, utilizing components such as DashboardForTransactions and DashboardForGroupTransactions. These components display charts for transactions to provide insights into user behavior related to managing their money.

## **Back End**

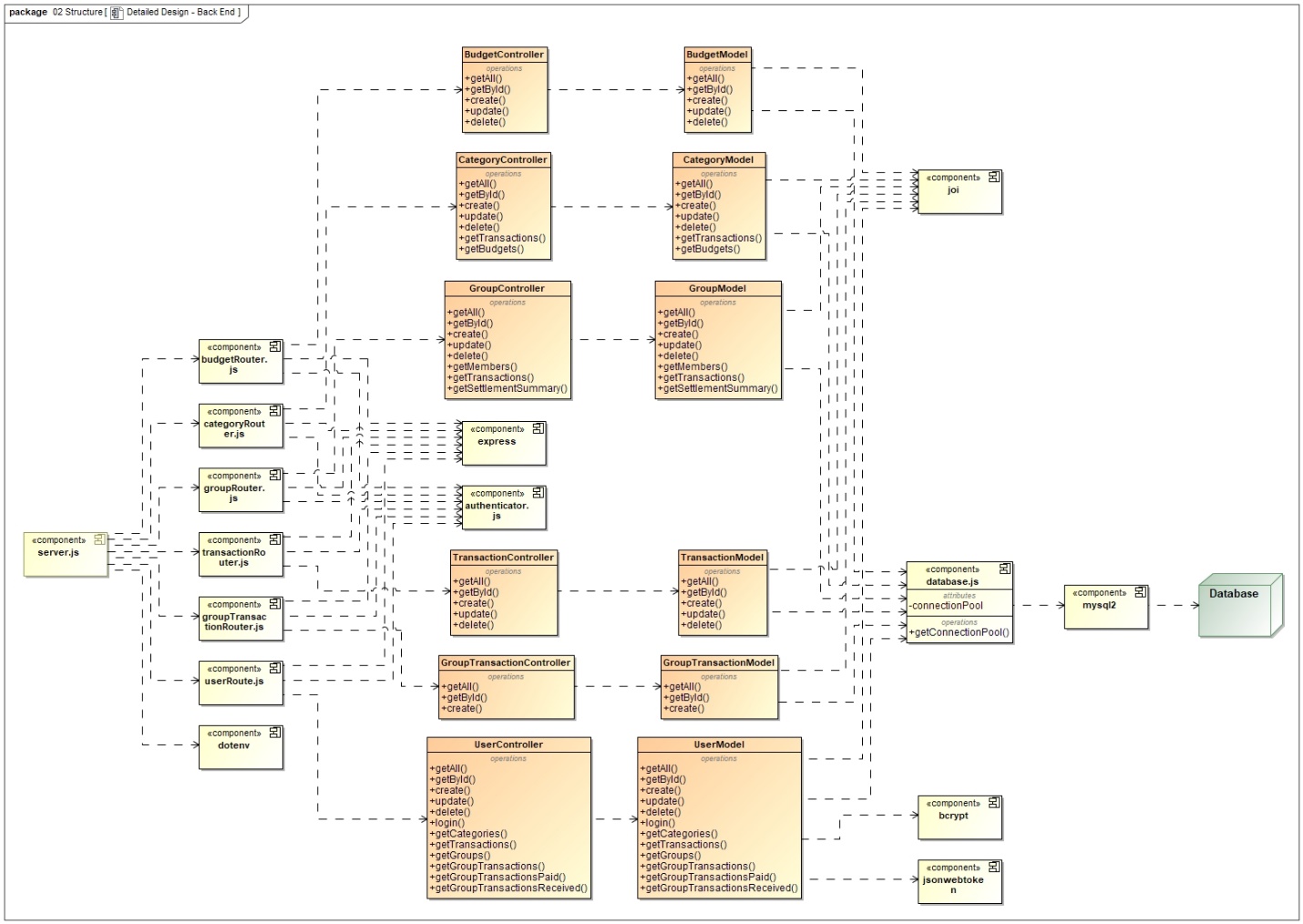


Figure Detailed design - Back End

Figure 20 presents a UML Component diagram detailing the backend design for the Expense Tracker application. Yellow entities represent UML Components, while orange entities represent UML Classes. The database is depicted as a green UML Node element.

Dependencies between classes, such as between BudgetController and BudgetModel, indicate that the controller class delegates its calls to the model to obtain results. This relationship is represented using UML Dependency relationships, signifying functional dependency rather than a Whole-Part (aggregation) relationship.

The backend process initiates by hosting a web server at a configured port, executed by the **server.js** component through the Node.js server. Server.js configures web service endpoints to corresponding processing router components.

**Router** components configure detailed endpoints to corresponding processing controller functions, incorporating authenticator middleware to ensure endpoints are protected and served only for authorized requests post-login.

**Controllers** manage HTTP request and response processing. Successful web service operations return 200 or 201 HTTP status codes to the client, while errors are communicated with a 500 HTTP status code.

Controllers utilize **model** components to communicate with the application database to fetch or modify data. This layered architecture facilitates separation of concerns and distribution of technical responsibility to each layer.

Several third-party modules are utilized for backend implementation:

* "**express**" implements the REST web service framework.
* "**joi**" validates client inputs.
* "**bcrypt**" and "**jsonwebtoken**" handle authentication and authorization.
* "**mysql2**" communicates with the MySQL database.
* "**dotenv**" loads configurations from the .env environment file.

Basic service patterns implemented by controllers and model components include:

* "**getAll**": GET HTTP request to retrieve all entities from the database.
* "**getById**": GET HTTP request, including entity identification, to retrieve entity details.
* "**create**": POST HTTP request, including entity attributes, to create the entity.
* "**update**": PUT HTTP request, including entity attributes and identifier, to modify the entity.
* "**delete**": DELETE HTTP request, including entity identifier, to delete the entity.

These classes serve as service classes and do not represent the entity itself, thus only hosting operations without attributes or state members. For the sake of simplicity of diagraming the parameters accepted by these service methods are also not shown here.

## **Activity Diagrams**

Here is an example scenario of the application to manage transaction Categories. There are four use cases here which are detailed subsequently using an activity diagram.

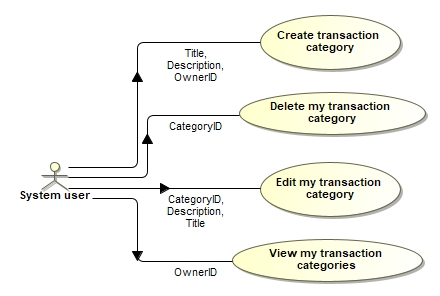


Figure Category management use cases

### **Use case: Create transaction category**

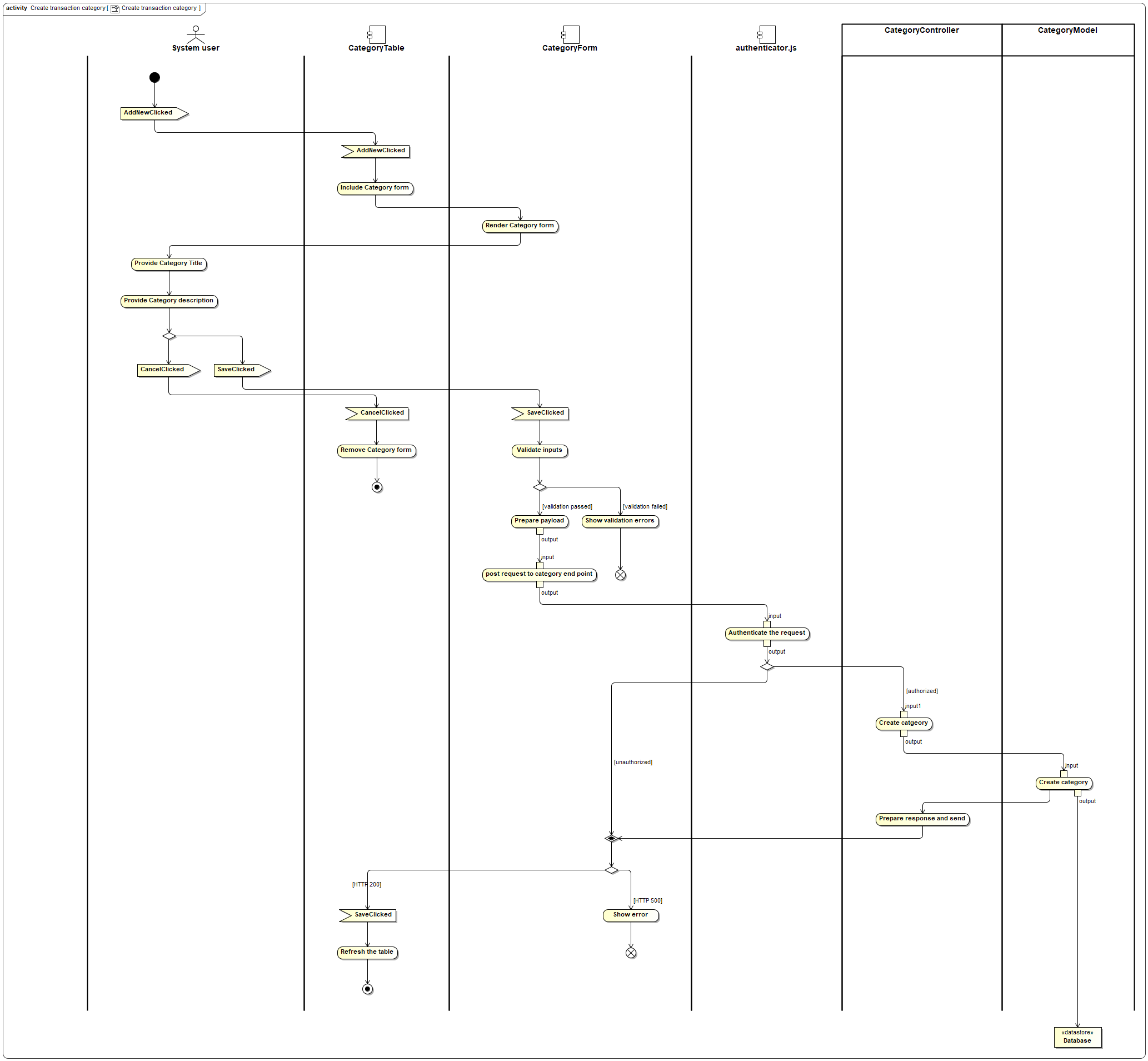


Figure Activity diagram - Create transaction category

As the activity diagram is very big and hard to read, please review the following description of it.

The activity diagram has vertical swim lanes, where each swim lane represent the component responsible for performing the corresponding actions in the activity diagram. The components representing these **vertical swim lanes** are

* System User
* CategoryTable
* CategoryForm
* authenticator.js
* CategoryController
* CategoryModel

Following symbols represents signals/event and receptions/event handlers in the activity diagram.

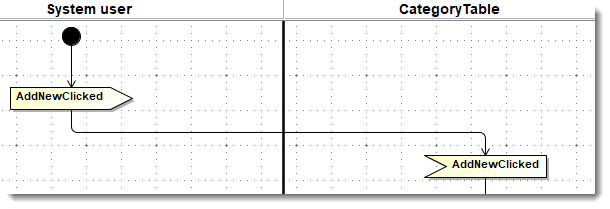


Figure Signals/Event and Receptions/Event handler symbols

The above diagram mentions that System User generate event AddNewClicked and CategoryTable component handles event AddNewClicked. So this diagram has the following **events and event handlers**:

* AddNewClicked
* CancelClicked
* SaveClicked

Following is the description of the activity diagram activities.

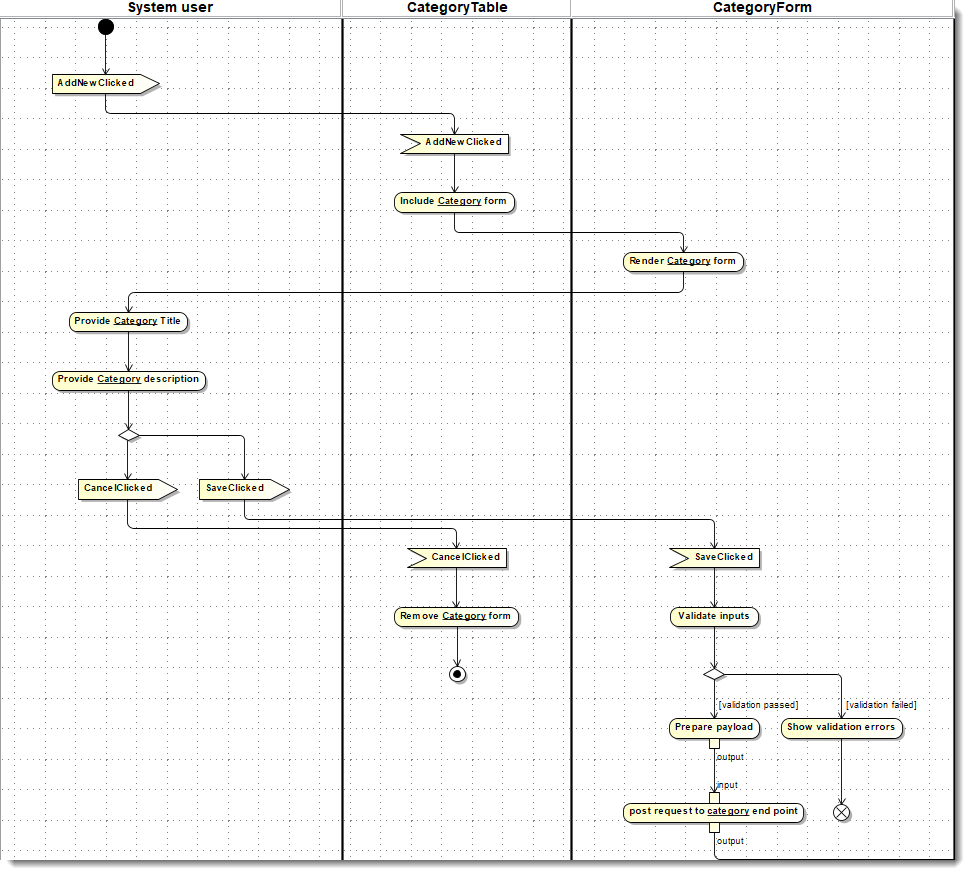


Figure Activity Diagram - Create transaction category - Part 1

* + - 1. System User
         1. Generates event AddNewClicked.
      2. CategoryTable
         1. Handles the event AddNewClicked.
         2. Includes the CategoryForm component.
      3. CategoryForm
         1. Renders the category form
      4. System User
         1. Provides category title on the form.
         2. Provides category description on the form.
         3. May now generate SaveClicked event OR generate CancelClicked event.
      5. CategoryTable
         1. Handles CancelClicked event by closing the CategoryForm.
      6. CategoryForm
         1. Handles SaveClicked event.
         2. Validates the input.
         3. If input validation is failed, then error is shown on the form and wait for System User at step 4.a
         4. If input validation is passed, prepares the payload and post the request to category web service end point.

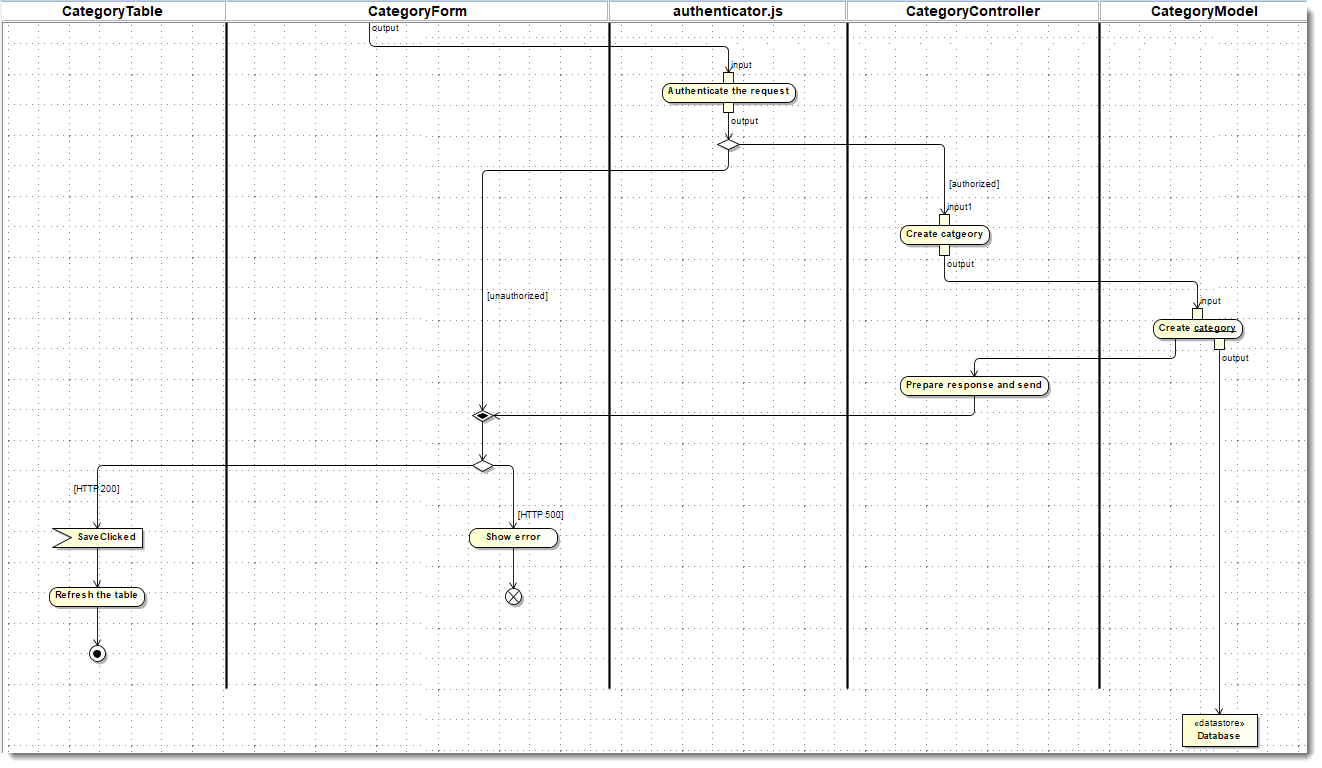


Figure Activity Diagram - Create transaction category - Part 2

* + - 1. authenticator.js
         1. Authenticates the request.
         2. If unauthorized request then report the error to client with HTTP 500 code.
         3. If authorized request, then pass on the payload to the CategoryController.
      2. CategoryController
         1. Accepts the payload and validates the data and then invokes methods passing the payload to CategoryModel.
      3. CategoryModel
         1. Accepts the payload and creates the category record in database.
         2. Accordingly the status of the operation is reported to the CategoryController.
      4. CategoryController
         1. If the CategoryModel operation is successful then prepares the HTTP 200 status code response.
         2. If the CategoryModel operation is unsuccessful then prepares the HTTP 500 status code response.
         3. This response is returned to the client CategoryForm.
      5. CategoryForm
         1. If response contains HTTP 500 code, then error is shown and the process waits at the System User providing input at step 4.a
         2. If response contains HTTP 200 code, then communicate to the parent component CategoryTable.
      6. CategoryTable
         1. On successful operation, the CategoryForm component is closed.
         2. The Category listing table is refreshed to show latest data from database.
         3. Activity is concluded.

### **Use case: Edit my transaction category**

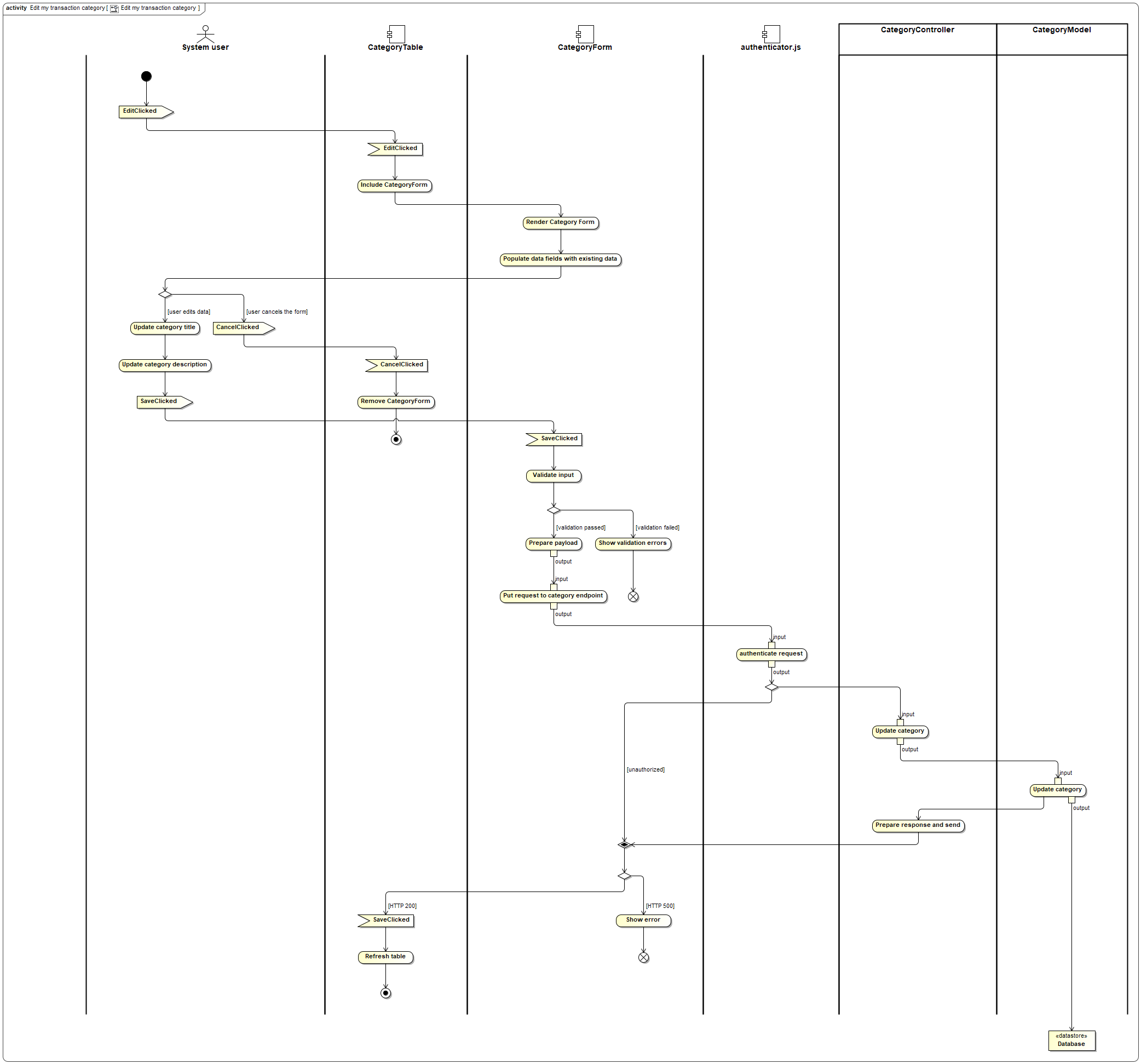


Figure Activity Diagram - Edit transaction category

Following is the description of the activity diagram activities.

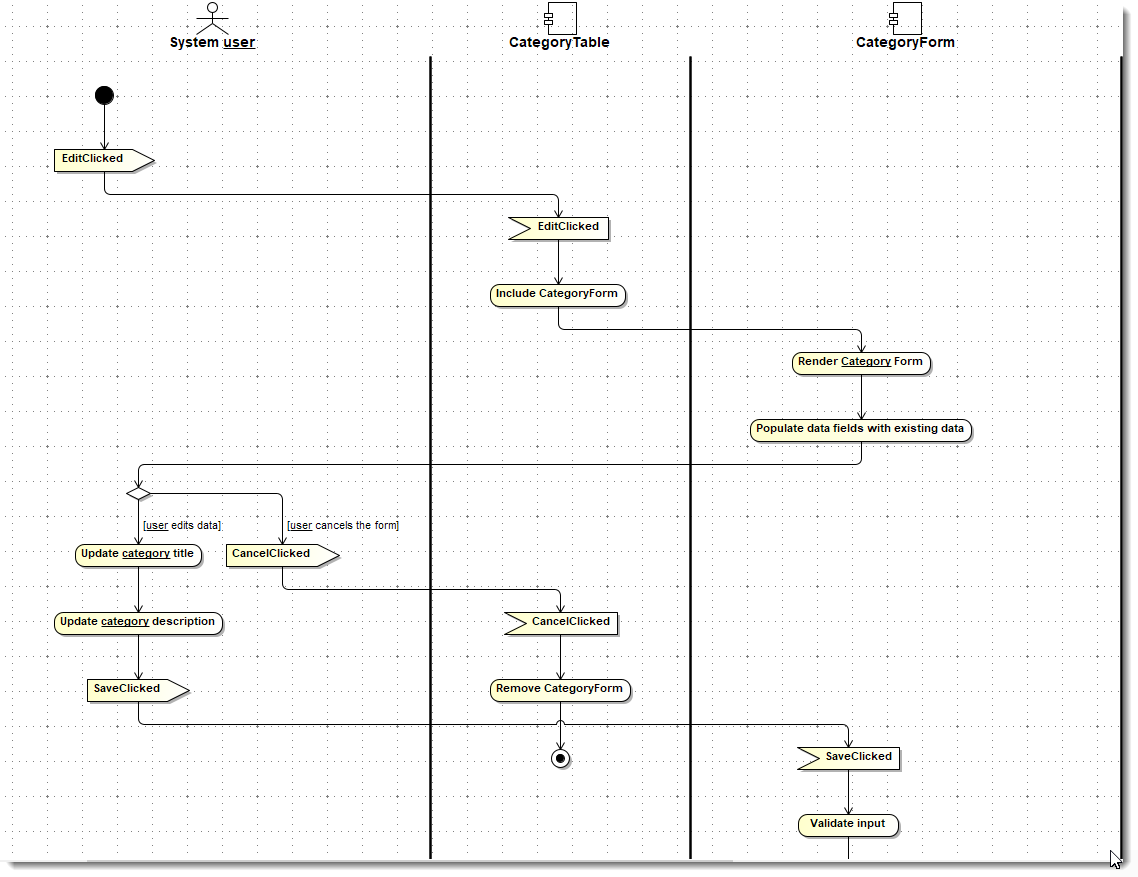


Figure Activity Diagram - Edit transaction category - Part 1

* + - 1. System User

Clicks and generates event EditClicked.

* + - 1. CategoryTable

Handles event EditClicked

Includes the CategoryForm for editing.

* + - 1. CategoryForm

Renders the category form

Populate data fields with existing data.

* + - 1. System User

May generate CancelClicked event, then it will handled by CategoryTable.

User updated the category Title and Description.

Clicks and generates event SaveClicked.

* + - 1. CategoryTable

If CancelClicked, then removes the CategoryForm.

* + - 1. CategoryForm

Handles SaveClicked event.

Validates the input.

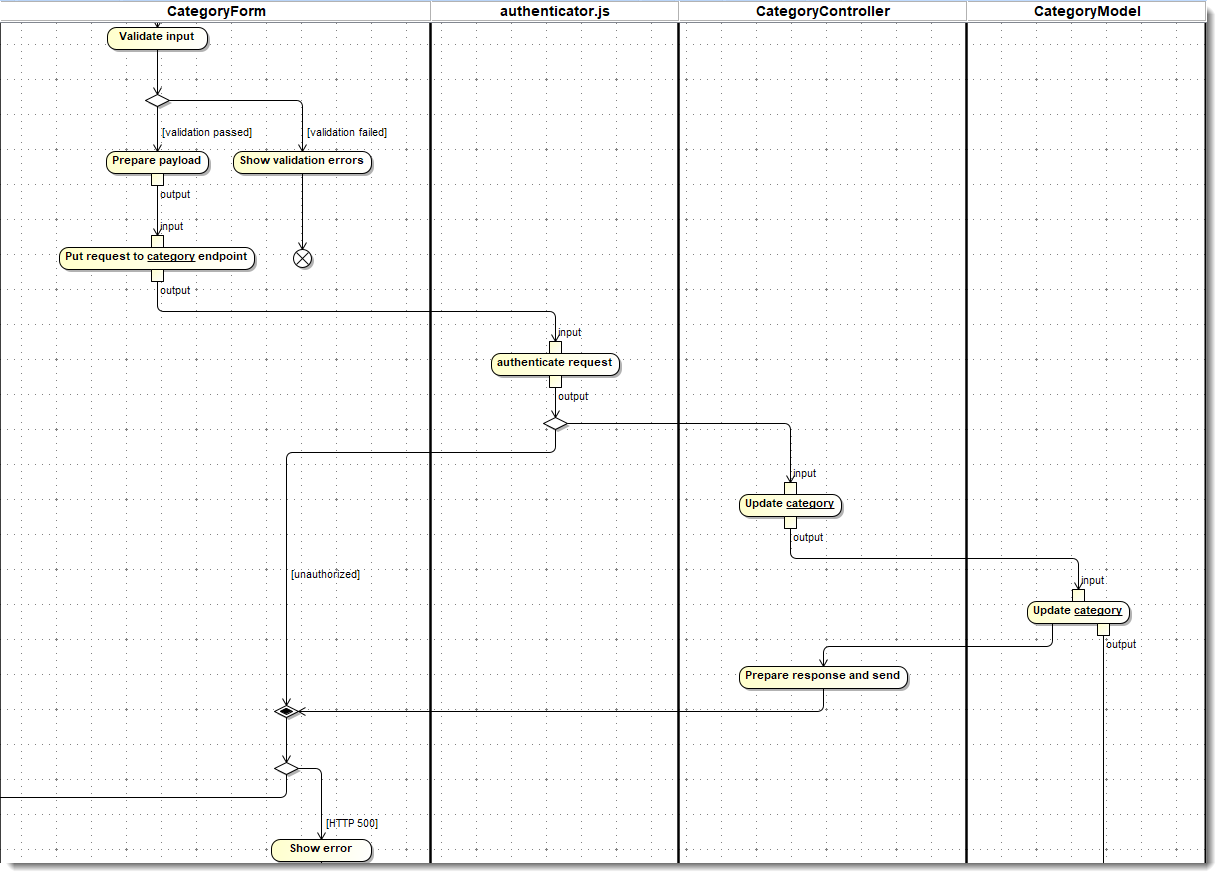


Figure Activity Diagram - Edit transaction category - Part 2

* + - 1. CategoryForm

If validation fails then shows validation error.

If validation passed then prepares the payload.

Sends put request to category endpoint.

* + - 1. authenticator.js

Authenticates incoming request.

If authentication unsuccessful then reports unauthorized error to CategoryForm.

If authentication successfully then delegates to category controller.

* + - 1. CategoryController

Delegate the request to category model.

* + - 1. CategoryModel

Update the category in database.

Reports the status of the operation to category controller.

* + - 1. CategoryController

Prepares the response and send it back to category form.

* + - 1. CategoryForm

If the response is HTTP 500 then shows the error

If the response is HTTP 200 then inform category table.

* + - 1. CategoryTable

Refreshes the table to show the latest data from database.

### **Use case: View my transaction categories**

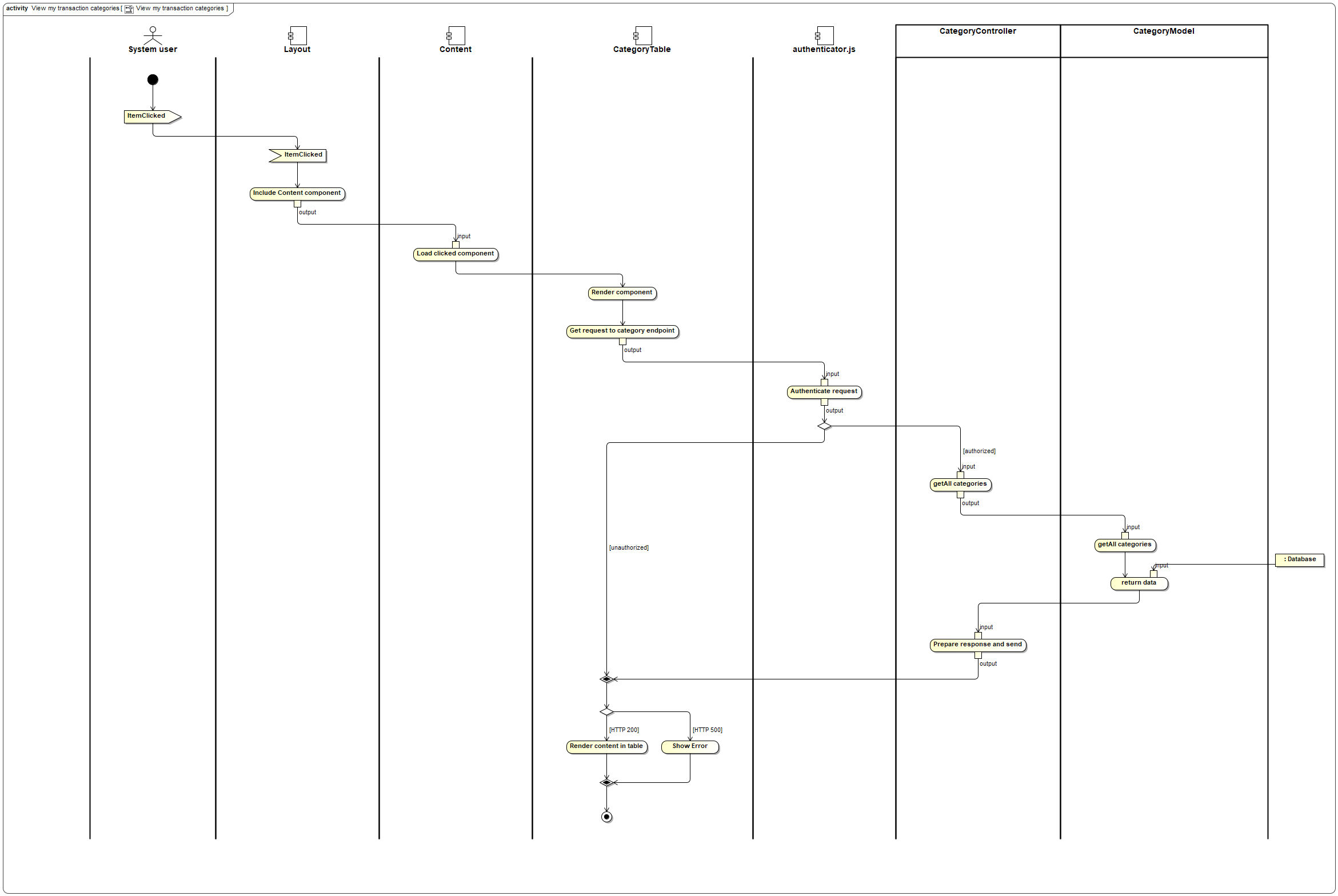


Figure Activity Diagram - View my transaction categories

Following is the description of the activity diagram activities.

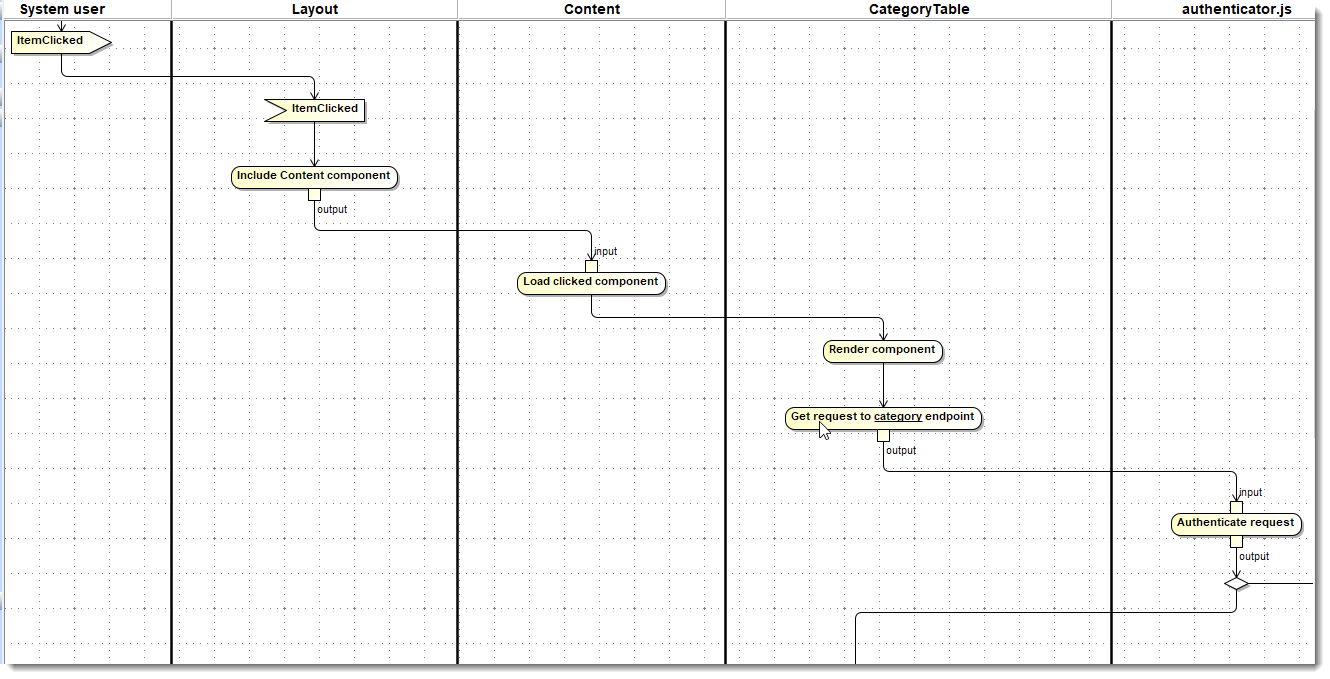


Figure Activity Diagram - View my transaction categories - Part 1

* + - 1. System User

Clicks and generates event ItemClicked.

* + - 1. Layout

Handles event ItemClicked

Includes the Content component.

* + - 1. Content

Load the clicked component i.e. CategoryTable

* + - 1. CategoryTable

Starts rendering the component.

Creates GET request to categories endpoint.

* + - 1. authenticator.js

Authenticates the request.

If authorized then request is passed to CategoryController.

If unauthorized then request is denied back to CategoryTable.

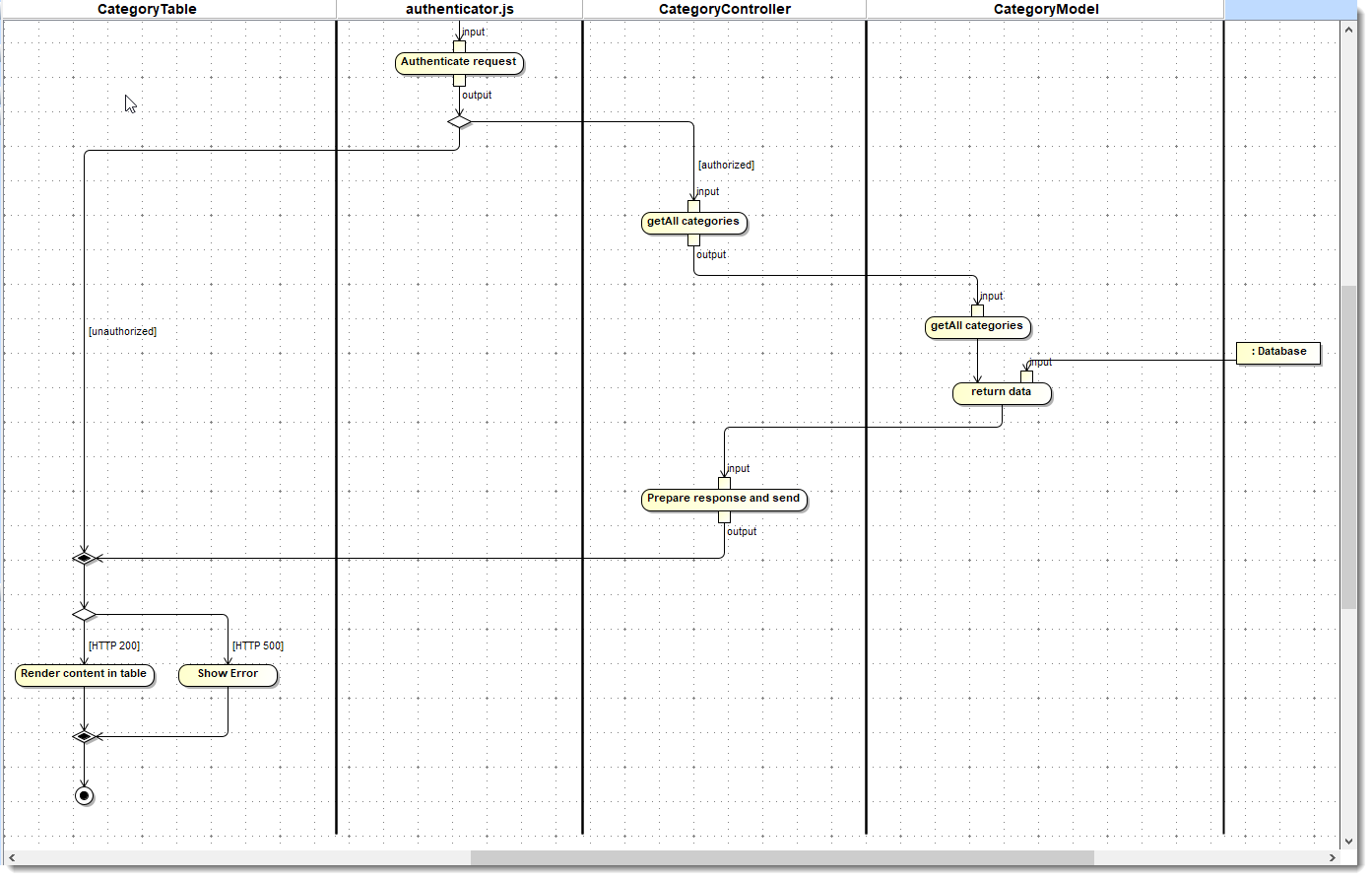


Figure Activity Diagram - View my transaction categories - Part 2

* + - 1. CategoryTable

If the unauthorized request then shows the error.

* + - 1. CategoryController

Prepares the input and delegates the getAll request to CategoryModel.

* + - 1. CategoryModel

getAll categories from database and return the data list.

* + - 1. CategoryController

Prepares the response and send back to client.

* + - 1. CategoryTable

If status code HTTP 200, then renders the data in table format.

### **Use case: Delete my transaction category**

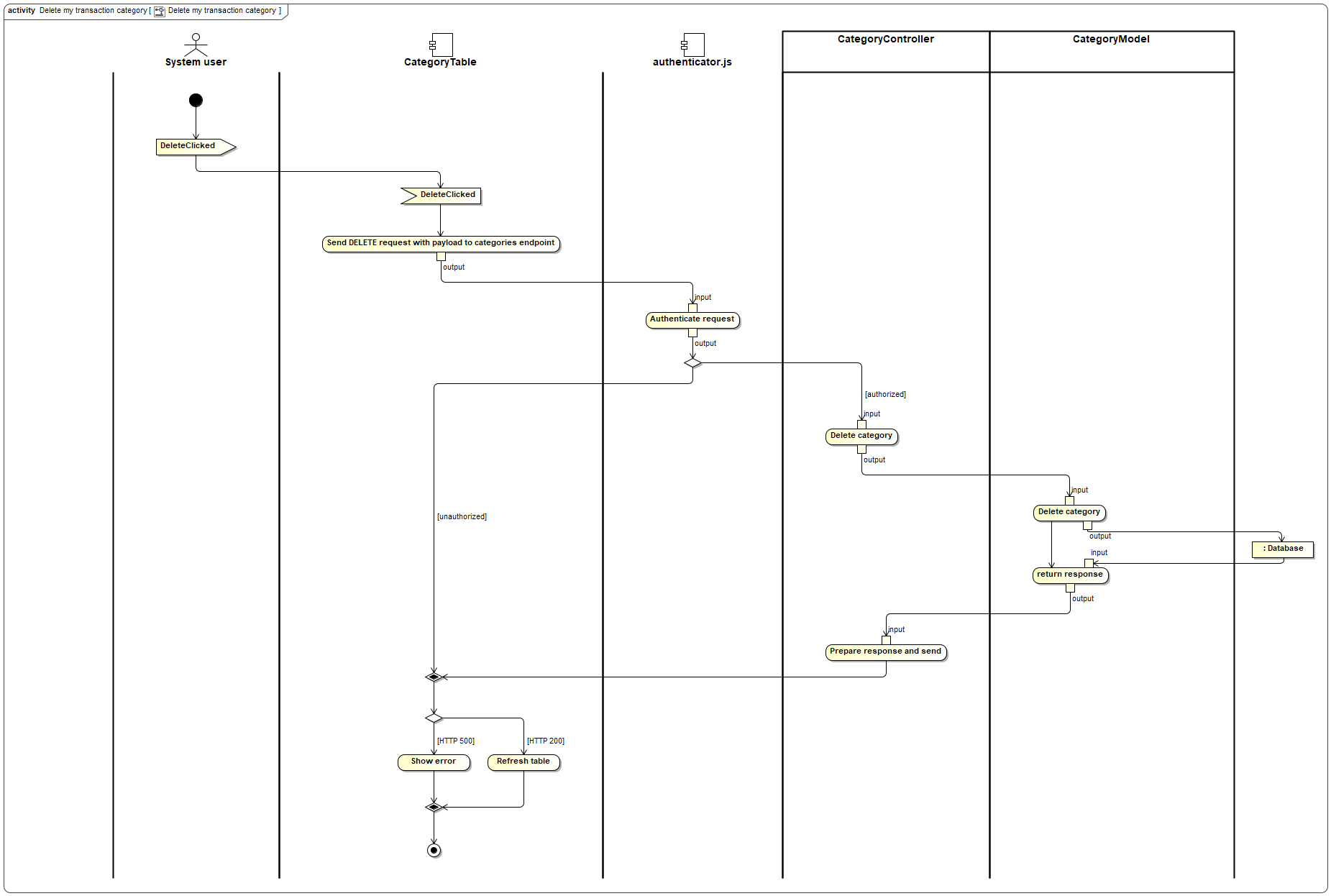


Figure Activity Diagram - Delete my transaction category

Following is the description of the activity diagram activities.

* + - 1. System User
         1. Clicks and generates event DeleteClicked
      2. CategoryTable
         1. Handles event DeleteClicked.
         2. Sends DELETE request with payload to categories endpoint.
      3. authenticator.js
         1. Authenticates request.
         2. If authorized then pass on to CategoryController.
         3. If unauthorized then request is denied back to CategoryTable.
      4. CategoryTable
         1. If request is denied then shows the error.
      5. CategoryController
         1. For the authorized request, prepares the input and delegates the Delete category to CategoryModel.
      6. CategoryModel
         1. Deletes the category is database.
         2. Prepares and returns response.
      7. CategoryTable
         1. For HTTP 200 response, refreshes the table to show latest data from database.
         2. For HTTP 500 response, shows the error.

# **Contributions**

|  |  |
| --- | --- |
| **Team member** | **Contribution** |
| Kapil Suryawanshi | Prepared draft of the design specification. |
| Aaradhana Sharma | Review and updates. |
| Suraj Odera | Review and updates. |
| Shayan Ali | Review and updates. |

# **Glossary**

|  |  |
| --- | --- |
| **Term** | **Description** |
| Budget | A predefined amount of money allocated for a specific purpose or period. |
| Category | A classification or group used to organize transactions based on their nature or purpose. |
| Cash In | Money received or earned as income or proceeds from a transaction. |
| Cash Out | Money spent or paid as an expense or cost for goods or services. |
| Group | A collection of individuals who share common interests or responsibilities. |
| Group Transactions | Financial transactions related to expenses shared among members of a group. |
| Monthly Overview Visualization | A graphical representation of financial data summarizing transactions over a month. |
| Status of Settlement | The current state or progress of settling financial obligations within a group. |
| Transaction | A financial exchange involving the transfer of money or assets between parties. |
| Transaction Entry | A record or entry in the system documenting a transaction. |
| User | An individual who interacts with the system to perform various tasks or functions. |
| External User | An individual who wants to interacts with the system to perform various tasks or functions, but does not have a registered account in the system yet. |
| System User | An individual who wants to interacts with the system to perform various tasks or functions, and has a registered account in the system. |
| Expense | Money spent or paid |
| Income | Money received or earned |
| Architecture | Design and structure of a system, including its components and their relationships. |
| Front End | The user interface and presentation layer of a software application. |
| Back End | The server-side of a web application, responsible for handling data and logic. |
| Database | A structured collection of data, typically stored and managed electronically. |
| Mockups | Visual representations of a user interface design, often static and non-functional. |
| Use Case | A description of how users interact with a system to accomplish specific goals. |
| Activity Diagram | A graphical representation of the flow of actions within a system or process. |
| Authentication | The process of verifying the identity of a user or system. |
| Authorization | The process of determining whether a user or system has permission to perform certain actions. |
| Routers | Components in a web framework responsible for mapping URLs to application actions. |
| Controllers | Components in a web framework responsible for handling user requests and invoking appropriate actions. |
| Models | Components in a web framework responsible for managing data and business logic. |
| Middleware | Software components that bridge different systems or components within a software application. |
| Module | A self-contained unit of functionality within a software system. |
| Layout | The arrangement and organization of elements within a user interface or web page. |
| ER Diagram | Entity-Relationship Diagram; a visual representation of entities and their relationships in a database. |
| Primary Key | A unique identifier for a record in a database table. |
| Foreign Key | A field in a database table that refers to the primary key in another table. |
| HTTP | Hypertext Transfer Protocol; the protocol used for transferring data over the web. |
| Swimlane | A visual element in diagrams, often used to represent responsibilities or roles in a process. |
| Event | An action or occurrence detected by a software system, often triggering responses or behaviors. |
| Event Handler | Code that responds to and handles events triggered within a software system. |

Table Glossary

# **References**

* + - 1. Project proposal - Project Proposal\_v4.docx
      2. Software Requirement Specification - " Software\_Requirement\_Specification\_v20240303-2.docx"
      3. Source code repository - <https://github.com/SuryawanshiKatUHV/COSC6342_ExpenseTracker/tree/Development>