[Banking APP]

Architecture/Design Document

**Table of Contents**

[**1**](#_gjdgxs) **INTRODUCTION 3**

[**2**](#_30j0zll) **DESIGN GOALS 4**

[**3**](#_1fob9te) **SYSTEM BEHAVIOR 4**

[**4**](#_3znysh7) **LOGICAL VIEW 5**

[**4.1**](#_2et92p0) **High-Level Design (Architecture) 5**

[**4.2**](#_3dy6vkm) **Mid-Level Design 6**

[**4.3**](#_1t3h5sf) **Detailed Class Design 8**

[**5**](#_4d34og8) **PROCESS VIEW 11**

[**6**](#_3rdcrjn) **DEVELOPMENT VIEW 12**

[**7**](#_2s8eyo1) **PHYSICAL VIEW 12**

[**8**](#_17dp8vu) **USE CASE VIEW 12**

Change History

**Version:** <1.0>

**Modifier:** Richard

**Date:** 10/27/2021

**Description of Change:** Adding the architecture and design requirements

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Version:** 1.0

**Modifier:** Ahmed

**Date:** 10/27/2021

**Description of Change:** Adding the architecture and design requirements

# **Introduction**

**Architecture and Design**

The purpose of the architecture/design document is to explain the organization of the code. A well-written architecture document will make it easier for new programmers to become familiar with the code.

The architecture/design document should identify major system components and describe their static attributes and dynamic patterns of interaction.

Software architecture and designs are typically expressed with a mix of UML models (class and sequence diagrams being the two most common) and prose. Dataflow diagrams are also helpful for understanding the interaction between components and overall flow of data through the system.

**About this Template**

This template suggests one way of documenting a software system’s architecture/design. You aren’t required to include every section in this template nor all the content in the sections you do include. However, the document you do submit should pass the following checklist:

* Are design objectives clearly stated? For example, if performance is more important than reusability, this should be made clear at the start of the design specification.
* Does the architecture partition the implementation into clearly defined subsystems or modules with well-defined interfaces?
* Does the architecture express in a clear way the main patterns of communication between subsystems and modules?
* Does the architecture satisfy the requirements?
* Is the architecture traceable to requirements?
* Any models created should either be expressed with a well-known modeling language, or if a well-known modeling language isn't used, the syntax and semantics of the symbols that are used should be defined.

This document describes the architecture and design for the <product name> application being developed for <customer>. <brief description of what the software does>.

The purpose of this document is to describe the architecture and design of the <Banking App> application in a way that addresses the interests and concerns of all major stakeholders. For this application the major stakeholders are:

* Users and the customer – they want assurances that the architecture will provide for system functionality and exhibit desirable non-functional quality requirements such as usability, reliability, etc.
* Developers – they want an architecture that will minimize complexity and development effort.
* Project Manager – the project manager is responsible for assigning tasks and coordinating development work. He or she wants an architecture that divides the system into components of roughly equal size and complexity that can be developed simultaneously with minimal dependencies. For this to happen, the modules need well-defined interfaces. Also, because most individuals specialize in a particular skill or technology, modules should be designed around specific expertise. For example, all UI logic might be encapsulated in one module. Another might have all business logic.
* Maintenance Programmers – they want assurance that the system will be easy to evolve and maintain on into the future.

The architecture and design for a software system is complex and individual stakeholders often have specialized interests. There is no one diagram or model that can easily express a system’s architecture and design. For this reason, software architecture and design is often presented in terms of multiple views or perspectives [IEEE Std. 1471]. Here the architecture of the <product name> application is described from 4 different perspectives [1995 Krutchen]:

1. Logical View – major components, their attributes and operations. This view also includes relationships between components and their interactions. When doing OO design, class diagrams and sequence diagrams are often used to express the logical view.
2. Process View – the threads of control and processes used to execute the operations identified in the logical view.
3. Development View – how system modules map to development organization.
4. Use Case View – the use case view is used to both motivate and validate design activity. At the start of design the requirements define the functional objectives for the design. Use cases are also used to validate suggested designs. It should be possible to walk through a use case scenario and follow the interaction between high-level components. The components should have all the necessary behavior to conceptually execute a use case.

# **Design Goals**

There is no absolute measure for distinguishing between good and bad design. The value of a design depends on stakeholder priorities. For example, depending on the circumstances, an efficient design might be better than a maintainable one, or vise versa. Therefore, before presenting a design it is good practice to state the design priorities. The design that is offered will be judged according to how well it satisfies the stated priorities.

The design priorities for the <product name> application are:

* The design should minimize complexity and development effort.
* The design should be <another design goal>.

# **System Behavior**

The use case view is used to both drive the design phase and validate the output of the design phase. The architecture description presented here starts with a review of the expected system behavior in order to set the stage for the architecture description that follows. For a more detailed account of software requirements, see the requirements document.

<brief description of system behavior>

# **Logical View**

The logical view describes the main functional components of the system. This includes modules, the static relationships between modules, and their dynamic patterns of interaction.

In this section the modules of the system are first expressed in terms of high level components (architecture) and progressively refined into more detailed components and eventually classes with specific attributes and operations.

## ***High-Level Design (Architecture)***

* The **Database** is the central storage location for all information on users, accounts, transactions, notifications, and settings
* The **API** is the set of defined functions that process data from the client or the database in order to carry out business logic.
* The **Security Middleware** controls authentication of the client's request and approving interactions between clients and the server.
* The **Client** is a web app built using react that interacts with the server and displays information to the user.

## ***Mid- and Low-Level Design***

<Explain and/or show static and dynamic aspects of subsystem components. Probably the most effective way of showing mid-level design is with class and sequence diagrams.>

Example Actions:

* Logging in

When logging, the user will navigate to the signin.html page. From there, they will type their email and password into the form. Upon submit of the form, a javascript function is called, which then takes the form data and posts it to the getToken API endpoint. The API searches the database for user salt and hash based on the email, and if it exists the password and salt are hashed. If both hashes match, a signed JWT token of the userId is returned to the client. If anything fails other than a database connection error (500), a 400 error is returned to the client. If successful, the client then stores the token in local storage and redirects the user; if not, the error is shown to the user (either "Incorrect email or password" or "Server error")

* Add transaction

Only admin accounts (human or nonhuman) can add transactions (this is a banking application, so why would we let the user deposit $500 unprompted). Upon call of the addTransaction API endpoint, the JWT middleware verifies the validity of the token and attaches the user to the request. The API then verifies that the user has permission to perform this action (confirm they are an admin). Once this is done, the API would look up the users for the account that the transaction is for. For each user, search the database for notifications that are enabled, and execute the functions that generate notifications on the transaction. After the notifications are generated, in an SQL transaction, add the transaction information and user notifications to the database. Error handling occurs here. Then either an error message is returned to the user, or updated account information (the balance and such).

Class Diagram - API Endpoints:

- `addAccount` \*\*[POST]\*\*

- NOTE: \*\*Requires Admin Access\*\*

- Parameters

- int `accountType`: The numerical Account Type to add

- list(int) `users`: List of UserIDs to add to account

- Returns: Account Information

```

{

"accountID": int,

"accountType": string,

"currentBalanceDollars": int,

"currentBalanceCents": int,

"users": [

{"userID": int,

"email": string,

"firstName": string,

"lastName": string}, ...

]

}

```

- `addAccountUser` \*\*[POST]\*\*

- NOTE: \*\*Requires Admin Access\*\*

- Parameters

- int `accountID`: The Account ID to modify

- list(int) `users`: List of UserIDs to add to account

- Returns: Updated Account Information

```

{

"accountID": int,

"accountType": string,

"currentBalanceDollars": int,

"currentBalanceCents": int,

"users": [

{"userID": int,

"email": string,

"firstName": string,

"lastName": string}, ...

]

}

```

- `addTransaction` \*\*[POST]\*\*

- NOTE: \*\*Requires Admin Access\*\*

- Parameters

- int `accountID`

- int `amountDollars`

- int `amountCents`

- int `timeMonth` (optional - default: today)

- int `timeDay` (optional - default: today)

- int `timeYear` (optional - default: today)

- string `locationStCd`

- string `countryCd`

- string `vendor`

- Returns: A transaction

```

{

"transactionID": int,

"timeMonth": int,

"timeDay": int,

"timeYear": int,

"amountDollars": int,

"amountCents": int,

"endBalanceDollars": int,

"endBalanceCents": int,

"locationStCd": string,

"countryCd": string,

"vendor": string,

"vendorDescription": string,

"vendorCategory": string

}

```

- `addTransfer` \*\*[POST]\*\*

- Parameters

- int `fromAccountID`

- int `toAccountID`

- int `amountDollars`

- int `amountCents`

- Returns: A transaction

```

{

"transactionID": int,

"timeMonth": int,

"timeDay": int,

"timeYear": int,

"amountDollars": int,

"amountCents": int,

"endBalanceDollars": int,

"endBalanceCents": int,

"locationStCd": string,

"countryCd": string,

"vendor": string,

"vendorDescription": string,

"vendorCategory": string

}

```

- `addUser` \*\*[POST]\*\*

- NOTE: \*\*Requires Admin Access\*\*

- Parameters

- string `email`

- string `firstName`

- string `lastName`

- string `addressLine1`: address line 1 of user

- string `addressLine2`: address line 2 of user

- string `city`: city of residence

- string `postalState`: two letter postal code of state

- Returns: User Information

```

{

"userID": int,

"email": string,

"firstName": string,

"lastName": string,

"SSN": int,

"addressLine1" string,

"addressLine2" string,

"city" string,

"postalState" string

}

```

- `deleteAccount` \*\*[POST]\*\*

- NOTE: \*\*Requires Admin Access\*\*

- Parameters

- int `accountID`: AccountID to remove

- Returns: Confirmation

- `deleteAccountUser` \*\*[POST]\*\*

- NOTEs:

- \*\*Requires Admin Access\*\*

- CANNOT remove all users

- Parameters

- int `accountID`: The Account ID to modify

- list(int) `users`: List of UserIDs to remove from the account

- Returns: Updated Account Information

```

{

"accountID": int,

"accountType": string,

"currentBalanceDollars": int,

"currentBalanceCents": int,

"users": [

{"userID": int,

"email": string,

"firstName": string,

"lastName": string}, ...

]

}

```

- `deleteUser` \*\*[POST]\*\*

- NOTEs:

- \*\*Requires Admin Access\*\*

- User CANNOT have an open account

- Parameters

- int `ID`: UserID to remove

- Returns: Confirmation

- `getAccount` \*\*[POST]\*\*

- Parameters

- int `ID`: The Account ID to get more information for

- Returns: Information about a account (email is masked)

```

{

"accountID": int,

"accountType": string,

"currentBalanceDollars": int,

"currentBalanceCents": int,

"users": [

{"userID": int,

"email": string,

"firstName": string,

"lastName": string}, ...

]

}

```

- `getAccount` \*\*[POST]\*\*

- Parameters

- int `Token`: The Account Token (generated from a user lookup) to get more information for

- Returns: Information about a account (email and account number is masked)

```

{

"accountID": string,

"accountType": string,

"currentBalanceDollars": int,

"currentBalanceCents": int,

"users": [

{"userID": int,

"email": string,

"firstName": string,

"lastName": string}, ...

]

}

```

- `getAccountSummary` \*\*[POST]\*\*

- Parameters

- int `token`: The Account Token (generated from a user lookup) to get more information for

- string `timePeriodType`: Type of time the period is in (`D`: Days, `M`: Months, `Y`: Years)

- int `timePeriod`: Length of time to generate report for

- Returns: Summary information about a account transactions over past time period

```

{

"accountID": int,

"accountType": string,

"currentBalanceDollars": int,

"currentBalanceCents": int,

"previousBalanceDollars": int,

"previousBalanceCents": int,

"categories": [

{"category": string,

"amountDollars": int,

"amountCents": int,

"vendors": [

{"vendor": string,

"amountDollars": int,

"amountCents": int}, ...

]}, ...

]

}

```

- `getTransaction` \*\*[POST]\*\*

- Parameters

- int `ID`: The Transaction ID to get more information for

- Returns: A transaction

```

{

"transactionID": int,

"timeMonth": int,

"timeDay": int,

"timeYear": int,

"amountDollars": int,

"amountCents": int,

"endBalanceDollars": int,

"endBalanceCents": int,

"locationStCd": string,

"countryCd": string,

"vendor": string,

"vendorDescription": string,

"vendorCategory": string

}

```

- `getTransactionHistory` \*\*[POST]\*\*

- Parameters

- int `ID`: The Account ID to get transaction history for

- int `pageSize`: The number of results to return per page

- int `pageNumber` (default = 0): The page of results you want to view

- Returns: List of transactions from newest to oldest

```

{

"Transactions": [

{"transactionID": int,

"timeMonth": int,

"timeDay": int,

"timeYear": int,

"amountDollars": int,

"amountCents": int,

"endBalanceDollars": int,

"endBalanceCents": int,

"vendor": string}, ...

]

}

```

- `getTransactionHistory` \*\*[POST]\*\*

- Parameters

- int `token`: The Account token to get transaction history for

- int `pageSize`: The number of results to return per page

- int `pageNumber` (default = 0): The page of results you want to view

- Returns: List of transactions from newest to oldest

```

{

"Transactions": [

{"transactionID": int,

"timeMonth": int,

"timeDay": int,

"timeYear": int,

"amountDollars": int,

"amountCents": int,

"endBalanceDollars": int,

"endBalanceCents": int,

"vendor": string}, ...

]

}

```

- `getUser` \*\*[POST]\*\*

- Parameters

- int `ID`: The User ID to get more information for

- Returns: Information about a user (with masked email and account numbers), and the account token can be used to get account information

```

{

"userID": int,

"email": string,

"firstName": string,

"lastName": string,

"accounts": [

{"accountID": string,

"accountType": string,

"accountToken": int,

"currentBalanceDollars": int,

"currentBalanceCents": int}, ...

]

}

```

- `getUserPrivileged` \*\*[POST]\*\*

- Parameters

- int `ID`: The User ID to get more information for

- Returns: Information about a user (with sensitive information and no masking)

```

{

"userID": int,

"email": string,

"firstName": string,

"lastName": string,

"SSN": int,

"addressLine1" string,

"addressLine2" string,

"city" string,

"postalState" string,

"accounts": [

{"accountID": string,

"accountType": string,

"currentBalanceDollars": int,

"currentBalanceCents": int,

"token": int}, ...

]

}

```

- `updateUser` \*\*[POST]\*\*

- NOTE: \*\*Requires Admin Access\*\*

- Parameters (all optional except UserID)

- int `ID`: UserID of user account you wish to update

- string `email`

- string `firstName`

- string `lastName`

- int `SSN`: user social security number

- string `addressLine1`: address line 1 of user

- string `addressLine2`: address line 2 of user

- string `city`: city of residence

- string `postalState`: two letter postal code of state

- Returns: Updated User Information (unmasked)

```

{

"userID": int,

"email": string,

"firstName": string,

"lastName": string,

"SSN": int,

"addressLine1" string,

"addressLine2" string,

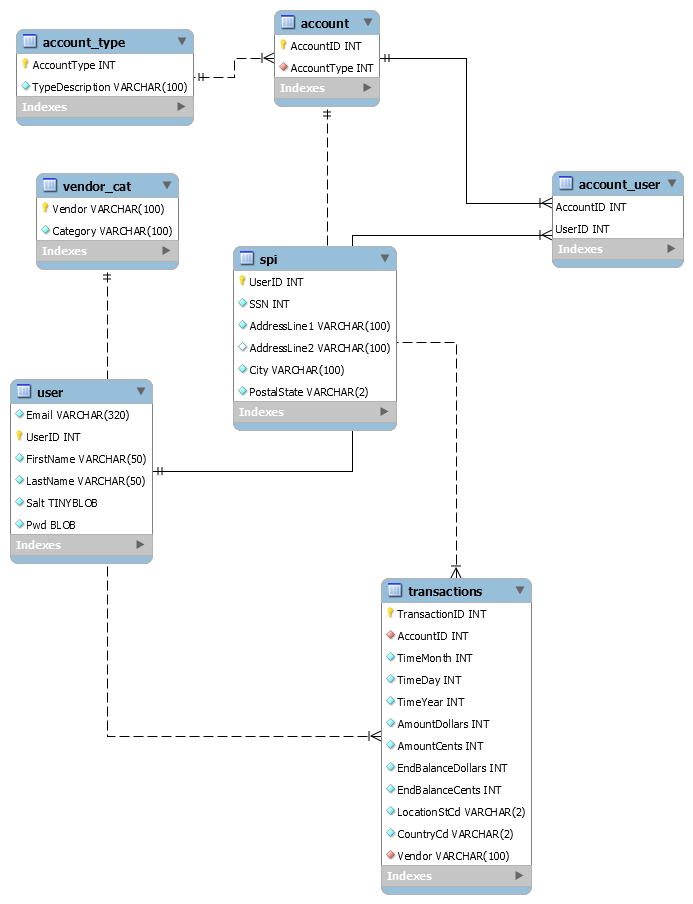
"city" string,

"postalState" string

}

```

Database:



# **Process View**

The application has a MySQL database server that runs and manages its own tasks. All communication occurs over internet connections.

The backend is a C# ASP.NET Core application that is run in an IIS Server. The server manages the worker processes, which can be scaled. All communication is to the API which is asynchronous, stateless, and over the internet.

The frontend is a React application that is hosted in development on a React-Scripts dev server, and for production, is built and hosted with Node.JS express server. Again, these interactions are all stateless and simply hosting.

The React client is in javascript and executed in the user's browser. This is also mainly single-threaded, but does make use of asynchronous methods to get and use data from the API.

# **Physical View**

MySQL database server - standalone on open port

C# ASP.NET Core - Proxied through load balancer and server Nginx

React application - Node.JS Express server connected to Nginx

# **Use Case View**

<Sketch architecturally significant use cases>

| **User** | **Client** | **Server** | **DB** |
| --- | --- | --- | --- |
| Fills out a form with credentials | Function gets form info  post from to getToken endpoint | Get user form data SQL search by email | Returns user email  salt and hash to server |
| clicks submit |  |  |  |
| 200: returns to page  400: flash message | 200: store token in local storage  400: notify user of incorrect credentials | Compute user hash w/ password and salt  verify password/hash match  if true: return JWT token  if false: 400 error |  |
|  |  |  |  |