Report for Software Quality and Process Management

Introduction.

An architecture in a software describes the components involved in it, how they communicate with each other. In our system (Gift List Service) components are the files that are encapsulating several different functionalities required by the system.

There are seven phases involved in software development life cycle, planning, Requirement gathering, Designing, Development, testing and maintenance. Architecture designing fall under Designing phase. There are several architectures designs available, layered, event driven, microkernel etc. and as developer we should have knowledge about these patterns, what benefits do each one them offer and what are their drawbacks so when the time arrives we can make the right decision.
Before explaining the architecture design pattern, I selected and my justifications for it, I would like to describe the problem and my approach towards it.

Planning.

In the first phase of my web application, I have clearly laid out the scope and aim of the application is to address a problem - During any event (wedding , birthday, anniversary) the person celebrating it receives many gifts from their guests, in some instances the gift they receive are repeated , also the guests at times gives a gift which the host is not interested in or not useful for them.
To address this issue, I have decided to develop a website where the host can create an event, add gifts or items they are interested in, and the users can then pledge a gift they will give to the host, this way the host will receive useful items, also which is not repeated again as other users will be able to see this when a pledge has been made.

Requirement gathering.

After planning and laying out the aim, I have assessed the information I will be requiring from different stakeholders of the system(users , guests).

- For any user to upload an event they will need to create an account in the system, once an account has been created they will be able to create an event with the following information :-
    - Event title , Event description , Event date , Image for the event.
    - Items for the event - item name, item link, item price.
- For any guest visiting the site no information will be required from them, they will be able to see the home page which contains each event title , image and date.

The UML case diagram illustrated below describes each person's role within the system also their associated actions they perform.

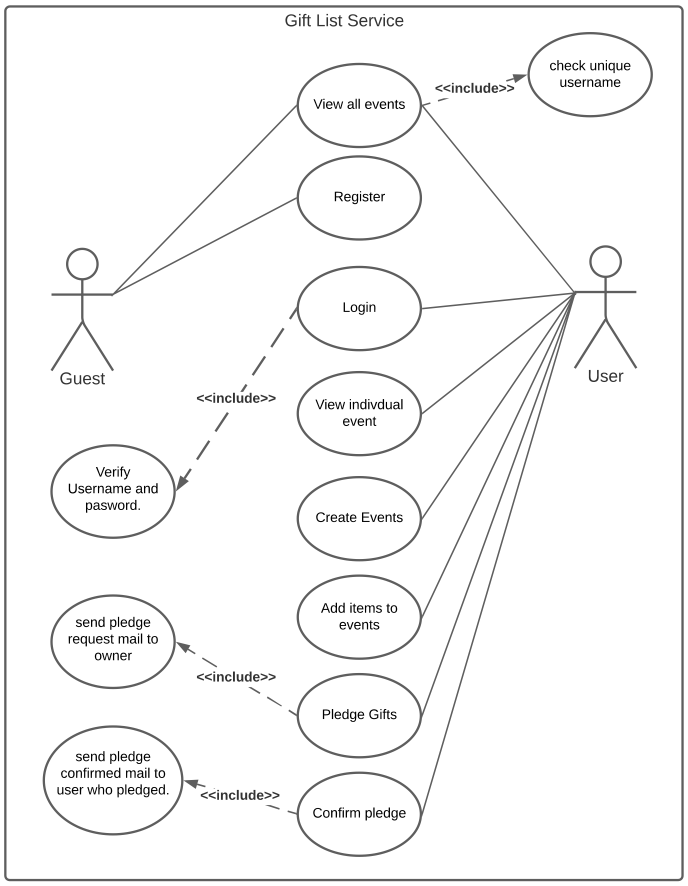


Figure 1 - UML case diagram

Design Architecture.

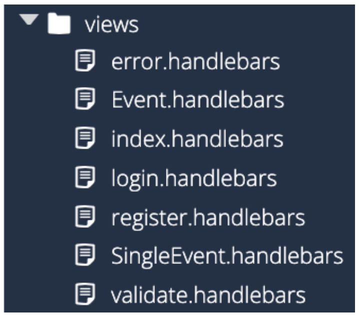
The design architecture I have followed is the layered architecture (also known as the n – tier architecture). We could add as many layers we like in this architecture, as there is no standard number of layers that need to be added. My system consists of the traditional layers or the most common layers present in a layered architecture which are the following : -

- Presentation layer
- Business logic layer
- Persistence layer
- Database layer

Let’s discuss each layer in the system, with examples.

Presentation Layer

Place where all our information is displayed to the user, UI interactions takes place and requests are originated.
I have created html files using html, CSS and JavaScript with a templating engine (handlebars) which will be used to display data appropriately receiving from the business logic layer of the system.

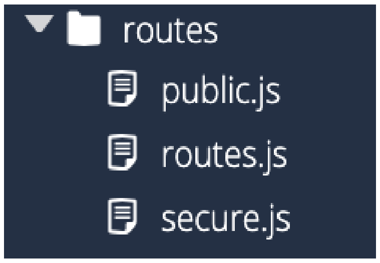


```
views
    error.handlebars
    Event.handlebars
    index.handlebars
    login.handlebars
    register.handlebars
    SingleEvent.handlebars
    validate.handlebars
```

Views folder in which all my html files are present. These html files act as the presentation layer of the system.

Business Layer

within this layer all our core business logic is present , upon which condition data is created, how they are processed, changed and stored in the database. It is the second layer where requests propagate after the presentation layer. Within my system the routes are the folder where all my requests are received and acts as the business logic layer.
Some of these routes are public and some of them are secured depending upon the functionality we would like to provide to the user depending upon their role (guest or user).



```
routes
    public.js
    routes.js
    secure.js
```

Public.js - the file where all our routes are public available.
Secure.js - the file where all our routes are secure(needs to register to access).
Routers.js - a router which nests all both these routers.

I have explained the entire business process with the help of a BPMN diagram below.

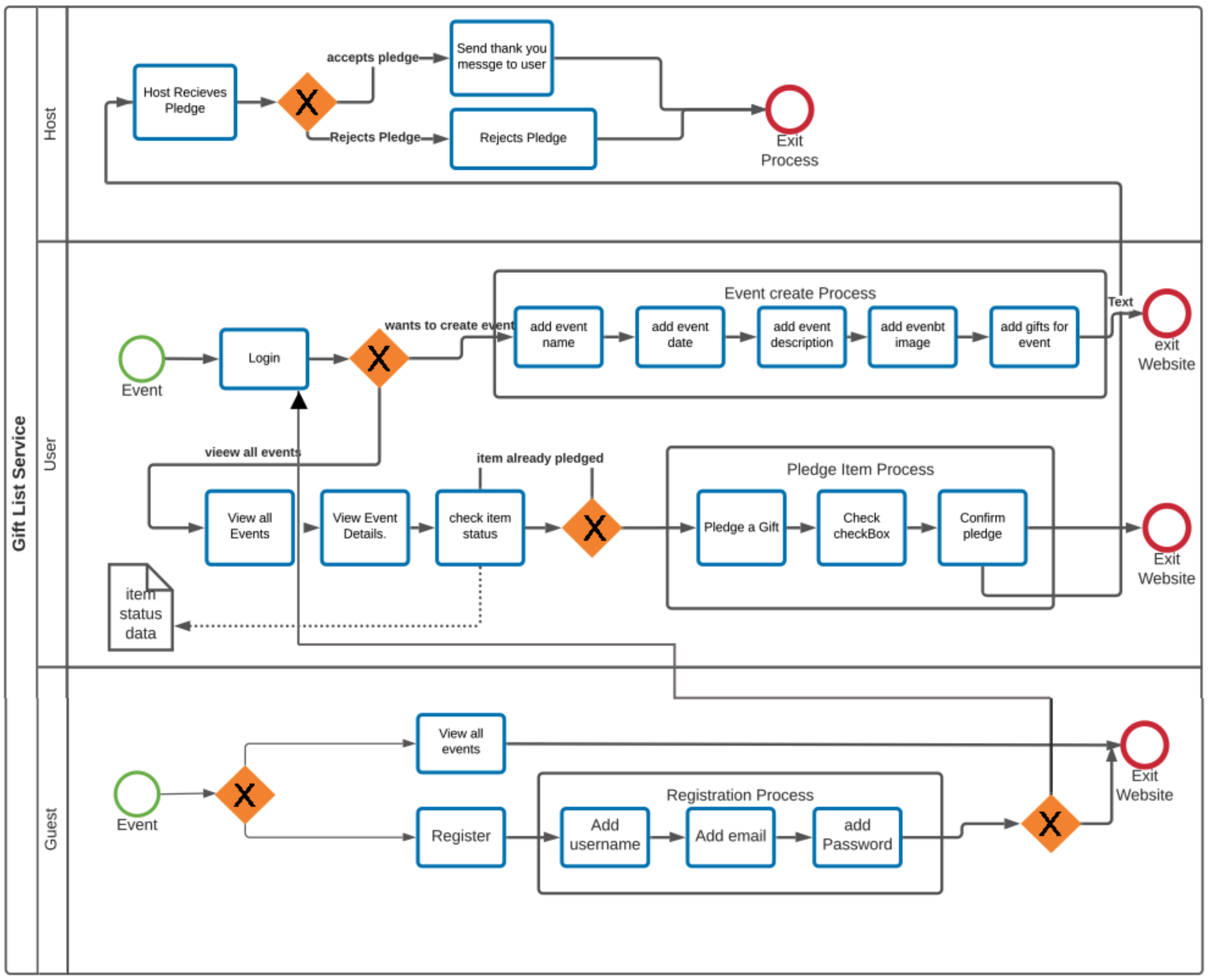
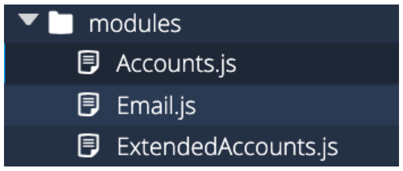


Figure 2 - BPMN diagram

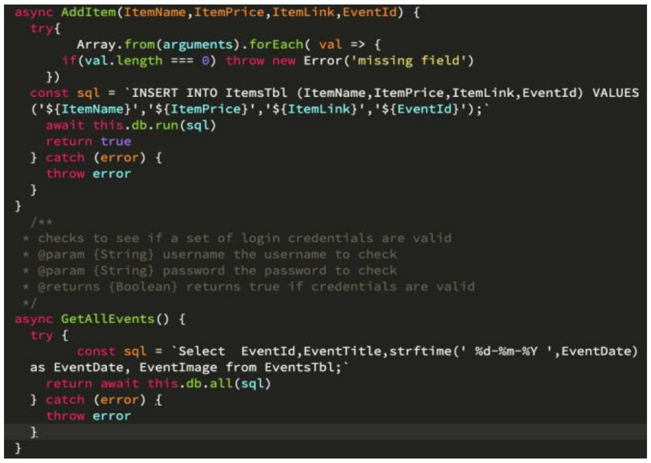
The user or the host have the same permissions both are provided with the same functionalities, we just recognize the user as a host when they host an event, and the user as person who is viewing their event page or pledging a gift for them.

Persistence layer

In a layered application architecture, the persistence layer is the place where we communicate with our database. The business layer makes a request for data through this layer. It contains data access objects which communicates with our database. It is the place where all our SQL queries are executed and we perform CRUD (create, read, update, delete) operation on our database as requested by the business layer.



In Gift list service, we have two files which become a part of this layer. Accounts and Extended Accounts



```
async AddItem(ItemName,ItemPrice,ItemLink,EventId) {
  try{
        Array.from(arguments).forEach( val => {
      if(val.length === 0) throw new Error('missing field')
    })
  const sql = `INSERT INTO ItemsTbl (ItemName,ItemPrice,ItemLink,EventId) VALUES
  ('${ItemName}','${ItemPrice}','${ItemLink}','${EventId}');`
    await this.db.run(sql)
    return true
  } catch (error) {
    throw error
  }
}
  /**
 * checks to see if a set of login credentials are valid
 * @param {String} username the username to check
 * @param {String} password the password to check
 * @returns {Boolean} returns true if credentials are valid
 */
async GetAllEvents() {
  try {
        const sql = `Select  EventId,EventTitle,strftime(' %d-%m-%Y ',EventDate)
  as EventDate, EventImage from EventsTbl;`
    return await this.db.all(sql)
  } catch (error) {
    throw error
  }
}
```

In this image we can see two function which are from Accounts.js file listed above, and so become a part of the persistence layer.

These functions are performing read and create operations on the database tables.

Database layer

For our database we have chosen SQLite. There are several factors which one could consider from Relationship's, Scalability, Performance, type and Size of data we would be storing. Since our application is not huge or has a complex design I have chosen SQLite it is easy to set up and operate, consists of a single file which can be stored in any directory and shared with others.

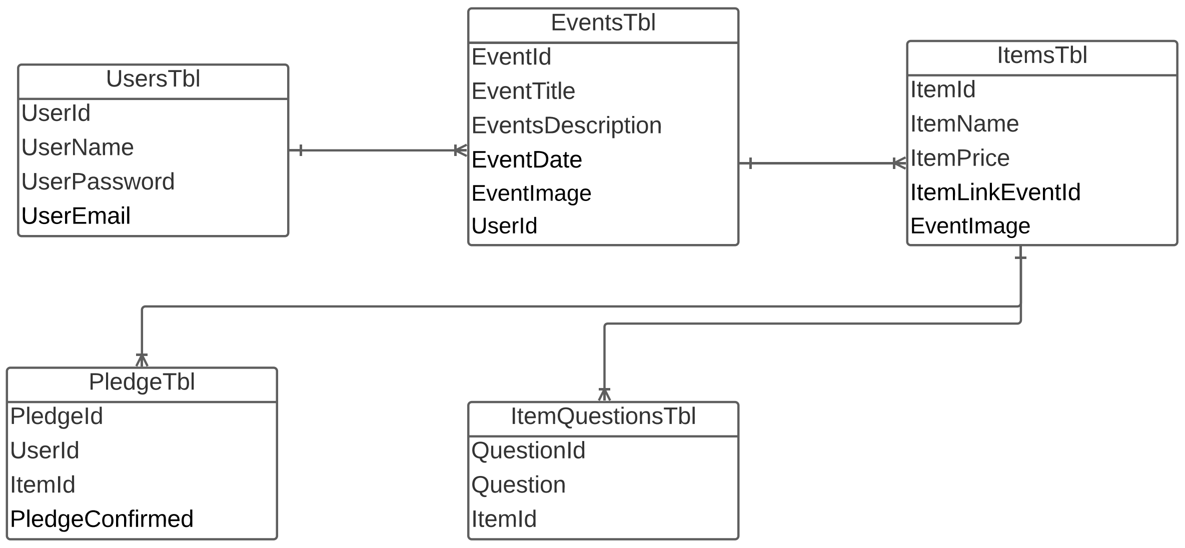


Figure 3 - ERD Diagram for Gift List service

- Each user on the system can create one or many events.
- Each event on the system can have one or many items linked to it.
- Each item can have more than one pledge made for it but will only contain one pledge confirmed by the host.
- Each item can have more than One or many questions posted for it.

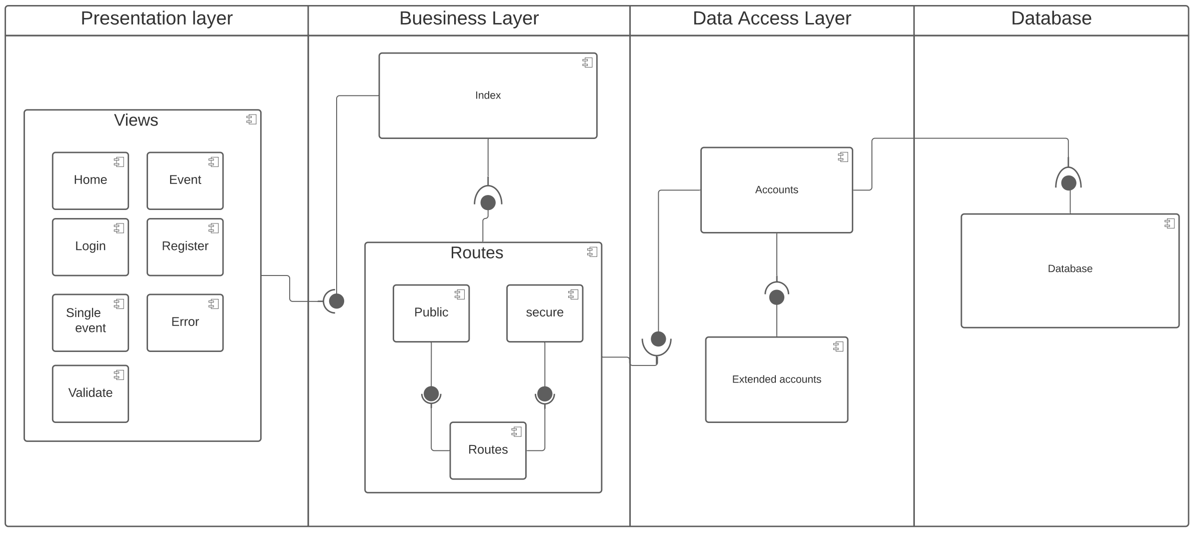


Figure 4 - Component Diagram

The diagram above represents the entire structure of the Gift list service web application. We can see the different components involved as files within the system and how they interact with each other.

Justifications for choosing Layered Architecture Pros and Limitations

Every developer considers certain pros and cons while choosing an architecture to follow. While developing I selected layered architecture for the following reasons : -

Pros

- Ease of Development : - this is the key reason to choose. layered architecture is a very common pattern among developers and one of the most used ones, even if a developer is not aware of design patters they might be using it intuitively. Most organization use this form of approach as they can separate the layers according to the skills set they have among developers. I was well aware of this pattern before learning about it and following the code as it grows is simple and easy.
- Reusable Components and layers of isolation: - Your components within the layers are very modular, we can use re-use these components (files) in any our other system or applications which saves time and man hours. Each layer works independently and performs certain given tasks only. They follow they concept of "layers of isolation" – which says changes made in one layer of the component don't affect the other layers.
- Testing : - We have performed several tests on our functions as seen in the image below. I have added all my files to be tested in one folder and testing suites in another. layered architecture makes testing different components in layers much simpler. As all our files are divided among layers mocking them, testing them is quite straightforward and easy to do. We can test all our functionalities in the business logic of our system.

Limitations

- Agility – as layers are tight coupled with each other, they will not be able to responds to change very quickly in a changing environment.
- Performance - layered architecture may not be the best design of choice for huge system, requests have to though several layers to respond back to the user, which could be time consuming and lower the performance of the system.
- Deployment - the nature of the application also determines the deployment process. Incorrectly implementing your pattern could cause issues in deployment. Detail changes could result in redeploying the entire application which would require planning weeks ahead.