# Gigster Interview Exam

**Note:** I have created two users

* User:
  + Username: [paraball@uncc.edu](mailto:paraball@uncc.edu)
  + Password: 33333333
* Admin:
  + Username: [araballi.pawan@gmail.com](mailto:araballi.pawan@gmail.com)
  + Password: 11111111

The github url : <https://github.com/pawanaraballi/ExpenseApplication>

## System Architecture

The application is designed with NodeJS and Express as server side application, AWS SQL as database and Front end as AngularJS

C:\Users\Pawan\AppData\Local\Microsoft\Windows\INetCacheContent.Word\systemArchitecture.png

## Website:

The technology stack for the website are as follows

* AnguarJS
* HTML5
* Javascript
* jQuery
* Bootstrap
* Xeditable
* Highcharts
* Moment.js

The website has various functionality handled. The functionalities implemented are

* Login page: To login to different users (Admin or user)
* User Dashboard: This page is the landing page after successful login of users. The page displays the expenses for the logged in user with pictorial representation of expenses incurred. You can add new expenses, delete expense, Edit expense and logout.
* Admin Dashboard: This page is the landing page after successful login of admins. The page displays the expenses for all the users with pictorial representation of expenses incurred. You can add new expenses, delete expense, Edit expense for any users and even logout.
* Logout: which is a util page to redirect it to the login page and remove the sessions.
* Error: This page facilitates to display errors if there is any error.

## Server:

The packages used for NodeJS Express application are as follows

* Express: To handle the server end using express
* Jsonwebtoken: Used for authentication purpose.
* Express-session: To handle the session in express application
* Body-parser: Parse incoming request bodies in a middleware before your handlers, available under the req.body property
* Morgan: HTTP request logger middleware for node.js
* Path: It allows to get the path of the directives
* Cookie-parser: Parse Cookie header and populate req.cookies with an object keyed by the cookie names. Optionally you may enable signed cookie support by passing a secret string, which assigns req.secret so it may be used by other middleware.
* Bcrypt: Used for hashing the user password using salt value.
* Mysql: Library to connect to MYSQL database
* Bookshelf: Used as an ORM for SQL interaction
* Knex: To establish the mysql connection
* EJS: Embeded Javascript

Login: The login functionality has been implemented

* /login – GET: The login functionality and opening the appropriate homepage for admin or user

Logout: The logout functionality has been implemented

* /logout – GET: The session will be removed

User: All the user functionality has been included in this file

* /user/home – GET: The landing page for the user.
* /user/addExpense – POST: Add new expenses for the user.
* /user/removeExpense – GET: Remove Expenses for the user.
* /user/updateExpense – GET: Update the expense with the username and timestamp
* /user/add – GET: Add new admin to the database

Admin: All the admin functionality has been included in this file

* /admin/home – GET: The landing page for the admin user.
* /admin/addExpense – POST: Add new expenses for the user.
* /admin/removeExpense – GET: Remove Expenses for any user.
* /admin/updateExpense – GET: Update the expense with the username and timestamp
* /admin/add – GET: Add new admin to the database

Database: For this assignment, I am using SQL Database from AWS. I have 4 tables:

* User – Stores the basic user details.
* Login – The login details which includes username and password is stored with the type of user.
* Expenses – All the expenses are stored with the username, description, amount and datetime
* Admin – Stores the basic admin details.

## Task Completed

* Multiple user accounts should be supported - Completed
* There should be two types of users: regular users and admins - Completed
* A regular user:
  + Can log in and log out - Completed
  + Can generate reports of their spending over time (described in more detail in a later requirement) - Completed
  + Can create, read, update, and delete (CRUD) expenses they own – Partial (Update not working UI Framwork issue but the backend code is written).
  + Cannot CRUD expenses they do not own - Completed
* An admin:
  + Should also satisfy requirements 3(a) through 3(c) - Completed
  + Can read all the saved expenses, including those which they do not own - Completed
  + Cannot create, update, or delete expenses they do not own - Completed
* An expense:
  + Is owned by exactly one user - Completed
  + Contains at least the following fields:
  + Datetime, the date and time the transaction was made - Completed
  + Amount, the amount of money (in USD, precision of 0.01) associated with the expense - Completed
  + Description, a string describing the details of th transaction - Completed
  + Can only be created by a logged in user - Completed
  + Can only be read by either the user who owns it or an admin - Completed
  + Can only be updated and deleted by the user who owns it - Completed
* A report:
  + Shows the total amount spent per week by the logged in user – Partial – Completed Monthly and Daily.
  + Can be filtered to only show expenses occuring within a user-provided datetime range
  + Can only be generated by users who are logged in - Completed
  + Should not contain expenses not owned by the user (even if the user is an admin) – Completed

Optional

* The application should be deployed to a publicly accessible place (e.g. Heroku) – Completed
* Users can change the report to aggregate spending per hour, day,month, and year (in addition to per week) - Incomplete
* Reports contain a time-series plot of the spending over time - Incomplete
* Multiple currencies for expense amounts are supported - Incomplete
* Automatically generated documentation (e.g. JSDoc for front-end, Swagger/RAML for back-end) Incomplete

## Security measures

I have taken under consideration of the basic security measures like

* The password being hidden
* Using JWT(JSON Web Tokens) for authentication
* Handling session
* Using bookshelf ORM to avoid SQL Injection
* Cross Origin error handling by making the server push the webpages.
* Everytime the token is validated before any operation is performed.

## Tests:

I have performed manual testing on all the modules implemented on this assignment as I was not aware of the tool PhantomJS and Casperjs.

**Testcases:**

**Unit Testing**

User/Admin login page

* Validation of Username
  + Validation of username to accept only valid email id and throw error for other inputs.
* Validation for password
  + Password should not be null.

User Homepage – Add Expense

* Username validation
  + Username validation is performed.
* Amount validation
  + Amount should only be numbers.
  + Precision of 2 decimal places.
* Timestamp validation
  + Date being converted properly from timestamp to yyyy/mm/dd hh/mm/ss
  + Time can be null as it would take the current timestamp

User Homepage - Graphs

* Graphs validation
  + Graphs updating on new addition of expense

User Homepage – Remove Expense

* Expenses removed from the database and the table after remove is clicked.
* Graphs are updated after the removal of expense.

Admin Homepage – Add Expense

* Username validation
  + Username validation is performed.
* Amount validation
  + Amount should only be numbers.
  + Precision of 2 decimal places.
* Timestamp validation
  + Date being converted properly from timestamp to yyyy/mm/dd hh/mm/ss
  + Time can be null as it would take the current timestamp

Admin Homepage - Graphs

* Graphs validation
  + Graphs updating on new addition of expense

Admin Homepage – Remove Expense

* Expenses removed from the database and the table after remove is clicked.
* Graphs are updated after the removal of expense.

**Integration Testing:**

Login Page:

* When the username and password has been filled with incorrect credentials
  + An incorrect error message would be displayed
* When the username and password has been filled with correct credentials
  + The user data is fetched and appropriate tag is collected
  + If the user is a user, then user dashboard page is loaded
  + If the user is an admin then the admin page is loaded.

Admin Homepage:

* When the admin is logged in
  + All the expenses from all the users has been fetched and then displayed
  + Only the admin’s expenses are editable
  + Only admin’s expenses can be deleted.
  + The Graphs only shows the data from the logged in admin
  + Data can be updated only with the logged in user

User Homepage:

* When the user is logged in
  + All the expenses from logged in users has been fetched and then displayed
  + Only the user’s expenses are editable
  + Only user’s expenses can be deleted.
  + Data can be updated only with the logged in user

## Road blockers faced

All:

* JWT login tokens
* Update functionality because of the x-editable framework
* Timestamp functionality because of the x-editable framework
* Hashing of the password using bcrypt or any of the hashing packages.
* Weekly graphs
* Unit testing using casperjs and phantomjs
* Automatically generated documentation tools Swagger and JSDoc

Still pending:

* Bcrypt: I was not able to perform this as I am using windows laptop and there is an issue with the python and Microsoft visual studio. It would work perfectly fine in MacOS but not implemented.
* Update functionality for xeditable
* Weekly graphs
* Automatically generated documentation tool Swagger and JSDoc

## Casper Testing

As I was unaware of the tool and because of the time constraints I could implement only the validation and testing. This particular test case would login with the credentials given and take a screenshot of the page after the action is performed.

var casper = require('casper').create({

pageSettings: {

loadImages: false,//The script is much faster when this field is set to false

loadPlugins: false,

userAgent: 'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.157 Safari/537.36'

}

});

//First step is to open login page

casper.start().thenOpen("http://localhost:3000/login", function() {

console.log("Login Page");

});

//Now we have to populate username and password, and submit the form

casper.then(function(){

console.log("Login using username and password");

this.evaluate(function(){

document.getElementById("inputEmail").value="paraball@uncc.edu";

document.getElementById("inputPassword").value="33333333";

document.getElementById("login").click();

});

});

//Wait to be redirected to the Home page, and then make a screenshot

casper.then(function(){

console.log("Make a screenshot and save it as AfterLogin.png");

this.capture('AfterLogin.png');

});

casper.run();

**Output:**

