

# Computer Science NEA Project

## Student Finance Tracker

An analytical tool for monitoring educational expenses, budgeting for students, and tracking financial goals.

Artem Gruzdev

### Contents

<b>PROJECT OVERVIEW .....</b>	<b>3</b>
<i>Scope of the Project .....</i>	<i>3</i>
<b>ANALYSIS .....</b>	<b>3</b>
<i>Survey .....</i>	<i>4</i>
<i>From an online survey I conducted among almost 29 college students, I gathered that: .....</i>	<i>4</i>
<i>Objectives .....</i>	<i>5</i>
<i>Interview to investigate the platform I should use. ....</i>	<i>6</i>
<b>DESIGN .....</b>	<b>7</b>
OVERALL .....	7
<i>Languages.....</i>	<i>7</i>
<i>Technology stack of website .....</i>	<i>7</i>
USER INTERFACE DESIGN .....	8
<i>Navigation bar .....</i>	<i>8</i>
<i>Expenses page .....</i>	<i>8</i>
<i>Income page .....</i>	<i>9</i>
<i>How subscriptions and salaries are monitored .....</i>	<i>10</i>
<i>Fund page .....</i>	<i>10</i>
<i>Goal page.....</i>	<i>11</i>
<i>Adding money to goal.....</i>	<i>12</i>
<i>Overview page.....</i>	<i>14</i>
<i>Database.....</i>	<i>14</i>
<i>Security .....</i>	<i>15</i>
<b>TECHNICAL SOLUTION .....</b>	<b>16</b>
GENERAL CLIENT-SERVER STRUCTURE .....	16
APP.JS .....	16
MODELS FOLDER (DATABASE).....	17
SECURITY .....	19
<i>Sign up .....</i>	<i>19</i>
<i>Log in .....</i>	<i>22</i>
USERCHECK.....	25
1-TIME INCOMES AND EXPENSES.....	26
<i>HTML for income expense page.....</i>	<i>26</i>
<i>Server side of exp/inc.....</i>	<i>28</i>

SUBS AND SALARIES ADDITION AND TRACKING .....	30
HTML SUBS AND CONSTANT INCOMES .....	30
SERVER SIDE OF SUBS/SALARIES ADDITION .....	32
<i>Post req handling</i> .....	32
<i>Delete req</i> .....	33
<i>Tracking of subs and salaries</i> .....	34
FUNDS PAGE .....	36
HTML FUNDS.....	36
<i>addFunds request</i> .....	40
<i>Withdraw post req</i> .....	41
<i>Delete funds req</i> .....	42
<i>Updatefunds req</i> .....	42
GOAL .....	44
HTML FOR GOAL .....	44
<i>Addgoals req</i> .....	49
<i>Add money to goal req</i> .....	50
<i>Delete goal req</i> .....	51
<i>Withdraw goal req</i> .....	51
<i>Hide/Show goal req</i> .....	52
<i>Calculate req</i> .....	52
<i>Other way to add money to the goal</i> .....	54
<i>Income page</i> .....	55
OVERVIEW PAGE SORTING .....	56
<i>Overview get</i> .....	56
<b>TESTING.....</b>	<b>58</b>
SECURITY TESTING .....	58
INCOMES AND EXPENSES TESTING .....	59
GOAL PAGE TESTING .....	59
OVERVIEW PAGE TESTING .....	59
<b>EVALUATION .....</b>	<b>59</b>
DID I MEET THE OBJECTIVES? .....	60
<i>Objectives</i> .....	60
<i>Comments:</i> .....	60
FEEDBACK FROM USERS .....	61
<i>Evaluation of feedback</i> .....	61

# Project Overview

The Student Finance Tracker Website is a platform designed to offer students a user-friendly environment for managing their financial activities. With features that cater to user authentication, tracking of expenses and income, subscription management, and financial goal setting.

## Scope of the Project

The project encompasses the creation of a comprehensive web-based finance tracking system with the following functionalities:

- **User Account Management:** Robust login and registration processes incorporating cookie-based session management and encrypted password storage.
- **Management of financial records:** The system enables you to add, amend and remove financial transactions that are either one-off or continuous.
- **Target and fund setting as well as monitoring:** They encompass the functionality through which users can set financial goals and keep track of their completion.
- **Data visualization and presentation:** Interactive charts and reports illustrate the financial behaviour of individuals, as well as their general economic sustainability.

## Analysis

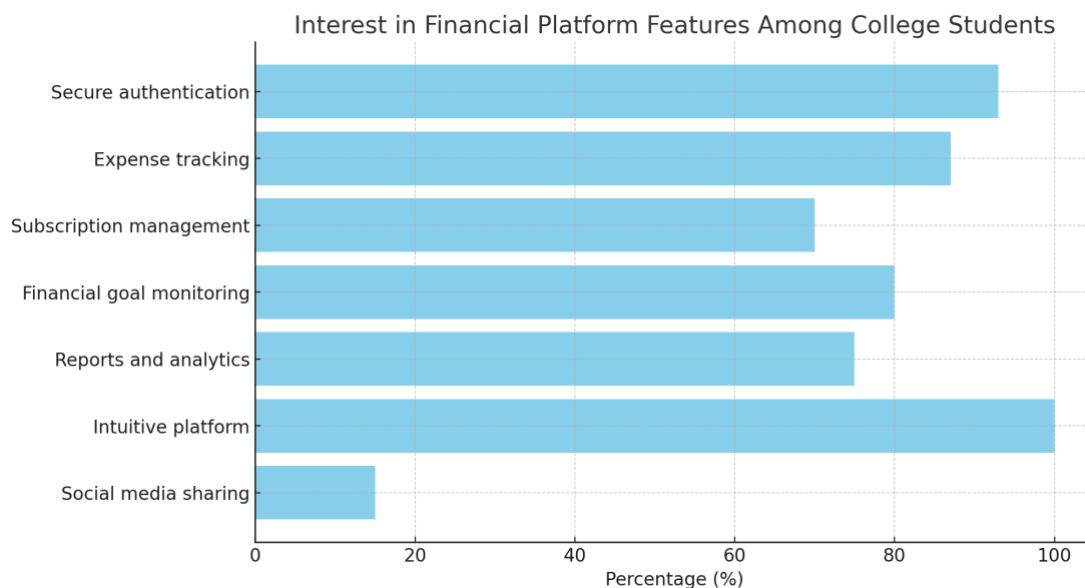
During my Student Finance Tracker Website project's analysis phase, I plan to address a particular issue encountered by students: managing one's finances across their academic **life**. Since the ability to understand how money works as well as control it is important for students who want to pursue studies and keep up with their friends in college, this finance tracker was designed with the aim of providing them with a tool which will aid them in budgeting and planning financially.

## Survey

**From an online survey I conducted among almost 29 college students, I gathered that:**

- The study found that 93% of the participants need a secure and easy way of authentication
- 87% showed interest in tracking daily expenses and incomes.
- 70% of people would like to have subscriptions management and ability to add salary
- 80% wanted to have a goal setting possibility as well as visualisation of the goal
- 75% wanted to get monthly reports and analytics rating their financial state
- 100% agreed on the platform to be as simple as intuitive as possible without any extra details
- Only 15% showed interest in sharing their financial insights on social medias

Visualisation of data:



From this investigation I found a specific functionality I should focus on and exact objectives for my project

## Objectives

I have incorporated these findings into the following project objectives:

1. A security system that allows for easy user authentication through password protection will be employed. Additionally, session cookies can be encrypted using hashing algorithms as well, thereby ensuring better protection from unauthorized access or any other security breaches while browsing the web.
2. Users should be able to update their data without experiencing any difficulties whatsoever. Thus, keeping all their income and expenses adding easy, quick, and simple becomes essential.
3. The site must be developed such that it manages subscriptions including periodic payments; hence there must be some features on it that remind users about their bills before the day of payment arrives especially for those who cannot remember anything at all about this.
4. Peer-oriented financial goal will find a place on this app to guide my course mates by setting your own targets. You can then follow your progress using an interactive chart animation.
5. We will generate personalized reports, that offer a clear picture of financial health to the individuals who wanted deeper insight into their finances, through a system of reporting and analytics.
6. In response to the wish for simplicity that was from all users, an interface that is very user-friendly remains at the top of my agenda.

## **Interview to investigate the platform I should use.**

**Objective:** The aim of this survey was to determine whether potential users prefer using a website or a mobile app for accessing digital content and services.

**Questions Included:** Preferences between websites and apps, reasons for their choice, frequency of use, and type of content accessed.

Results:

Total Participants: 25

Preference for Websites: 17 participants (68%)

Preference for Apps: 8 participants (32%)

### **Key Findings:**

#### **Reasons for Preferring Websites:**

1. Easier access without the need to install anything: 70% of website choosers
2. Better user experience on larger screens: 60% of website choosers
3. Perception of more comprehensive content available: 50% of website choosers

#### **Reasons for Preferring Apps:**

1. Convenience of accessing content on the go: 75% of app choosers
2. Faster loading times: 62.5% of app choosers
3. Better personalization options: 50% of app choosers

According to the data gathered from this interview, I stayed with developing a website to meet potential user wants.

# Design

The project is constructed with attention to detail across all user interface elements, backend services and the database structure. Here's how the key aspects of the solution are organized.

This project was created from scratch without using any templates, so I had to choose my tools and languages to use very careful.

## Overall

### Languages

I used JavaScript for writing scripts and main parts of code (backend and frontend) as I feel comfortable with this language and it allows use of async functions which is needed in web development.

HTML and EJS I used for frontend

CSS was used to create a smart and modern style of my website

### Technology stack of website

- Regarding the Frontend, it uses EJS, Embedded JavaScript Templates, to input dynamic data into HTML.
- Concerning the Backend, it uses a Node.js environment within a framework based on Express.js.
- For the storage of data it uses MongoDB, NoSQL database.
- To ensure user data security, it has cookie-based sessions and the passwords are hashed using bcrypt algorithm."

Explanation:

EJS (Embedded JavaScript Templates): Enables uncomplicated, server-side rendering of HTML with dynamic data.

Node.js and Express.js: Node.js offers a scalable environment for handling many connections, and Express.js simplifies server side logic and API management, thus increasing effectiveness.

MongoDB is a NoSQL database, it offers flexibility in terms of data storage and schema design which is great for developing data models. As well as that, it scales well and perform efficiently at large scales.

# User Interface Design

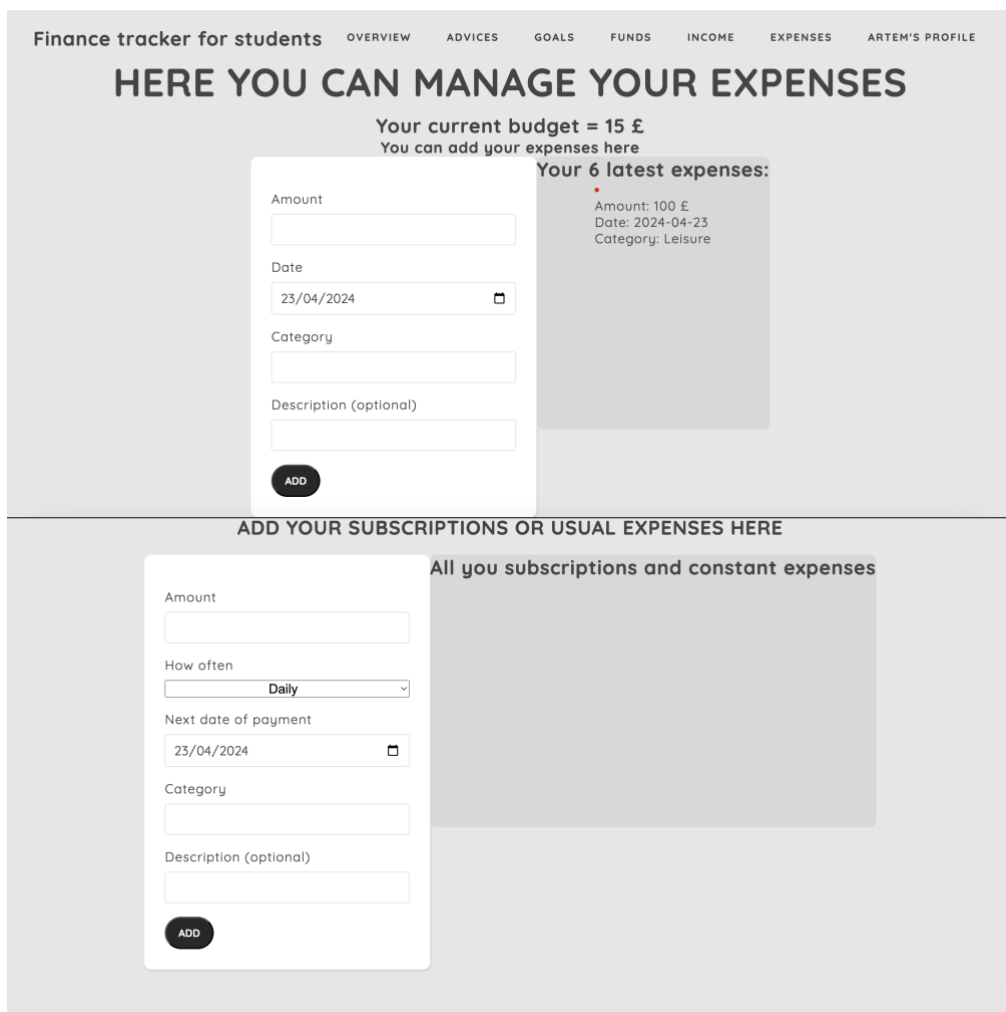
The website has a clear, intuitive navigation system that allows users to move smoothly between different financial management tools. With a focus on simplicity and usability, each feature is accessible through a main navigation bar, minimizing the learning curve for new users.

## Navigation bar



The design aims for clear and intuitive navigation through the website's financial management tools. The bar itself is placed at the top of the interface, facilitating easy access to the various features from a single point.

## Expenses page





This page has one-time expenditure adding option and of course subscription monitoring with ability to add new ones which is incredibly important thing to keep track of.

Both forms have amount to enter, date, category, and optional description. Subscription form also asks for frequency of the subscription.

By pressing “ADD” if all the required fields are filled, request will be sent to backend to add push new object either into expenses array or subscriptions array of user object.

## Income page

Finance tracker for students   OVERVIEW   ADVICES   GOALS   FUNDS   INCOME   EXPENSES   ARTEM'S PROFILE

## HERE YOU CAN MANAGE YOUR INCOMES

Your current budget = 30 £  
You can add your income here

Amount

Date

Category

Description (optional)

ADD

### Your 6 latest incomes:

- Amount: 15 £  
Date: 2024-04-23  
Category: Saved cash

### ADD YOUR SALARY OR USUAL PROFIT HERE:

All your constant income sources so far

Amount

How often

Next date of payment

Category

Description (optional)

ADD

This page is similar to expense page but for incomes and salaries.

Same forms and same way of pushing new object into income and salaries arrays in the user object.

## How subscriptions and salaries are monitored

When sub or salary is added **user.updated** becomes true, with the next authentication check which happens every get request, subs and salaries are updated. If current data is more or equal to next date of payment, then new income or expense is added.

## Fund page

# HERE YOU CAN MANAGE YOUR FUNDS

You can set goals and create funds here

Your current budget: 1000 £

Create a fund here:

Amount to raise

Name

ADD

### Your current funds

- To raise: 100 £
- For a new computer and raised: 60

Withdraw fund

You can add or withdraw money from your funds here

The amount would be added or taken from the budget

Choose a fund to update

For a new computer

Choose a type

Add

Amount to add

ADD

Fund page also has 2 forms, first is to create a fund and second is to add money on fund or withdraw money from it. Money are taken from budget if adding and come back on budget if withdrawing. Also each fund has green bar showing how much is left for money to be raised.

## Goal page

Goal page if you don't have any subscriptions or constant sources of income. The reason is that there is an algorithm which suggests the amount to save per day according to money spent and gained from subscriptions and incomes

Finance tracker for students OVERVIEW ADVICES GOALS FUNDS INCOME EXPENSES ARTEM'S PROFILE

### HERE YOU CAN MANAGE YOUR GOALS

**How goals work?**  
Goal includes amount to save and the date until which you want to accomplish this goal.  
We will provide you with data and advices on how to reach desired

**Your current budget: 123 £**

To use goals you should have at least 1 subscription and 1 source of constant income

Appearance when you have just created a new goal.

### Your current goal

- Amount to save: 100 £
- Date until which to save: 2024-04-28

DELETE CALCULATE HIDE

Here is the goal page when you pressed "CALCULATE" button which appears when you create a goal.

You also can hide or show the bar at the top of the page as a reminder of the goal

Goal progress: 0 £ of 150 £

Finance tracker for students OVERVIEW ADVICES GOALS FUNDS INCOME EXPENSES ARTEM'S PROFILE

### HERE YOU CAN MANAGE YOUR GOALS

**How goals work?**  
Goal includes amount to save and the date until which you want to accomplish this goal.  
We will provide you with data and advices on how to reach desired

**Your current budget: -20 £**

To use goals you should have at least 1 subscription and 1 source of constant income

**Your current goal**

- Amount to save: 150 £
- Date until which to save: 2024-04-26

DELETE HIDE

We analysed your goal:  
To reach your goal you have to gain at least 50 £ each day  
This is approximate calculation without including your expenses

**Amount saved: 0 £**

Amount to add to goal

ADD

Goal object is added into goal array for an easier reach later.

## Adding money to goal

As soon as goal appears, income page changes.

The screenshot displays a user interface for managing a budget. At the top, it states "Your current budget = 1000 £" and "You can add your income here". A modal form is open in the center, allowing users to add income. The form includes fields for "Amount", "Date" (pre-filled with "25/04/2024"), "Category", "Description (optional)", and "Percentage of income to goal (optional)". An "ADD" button is at the bottom of the modal. To the right, a section titled "Your 6 latest incomes:" shows a list of recent income entries, with the first entry displayed: "Amount: 123 £", "Date: 2024-04-25", and "Category: 123".

**Your current budget = 1000 £**  
You can add your income here

**Your 6 latest incomes:**

- Amount: 123 £  
Date: 2024-04-25  
Category: 123

**Form Fields:**

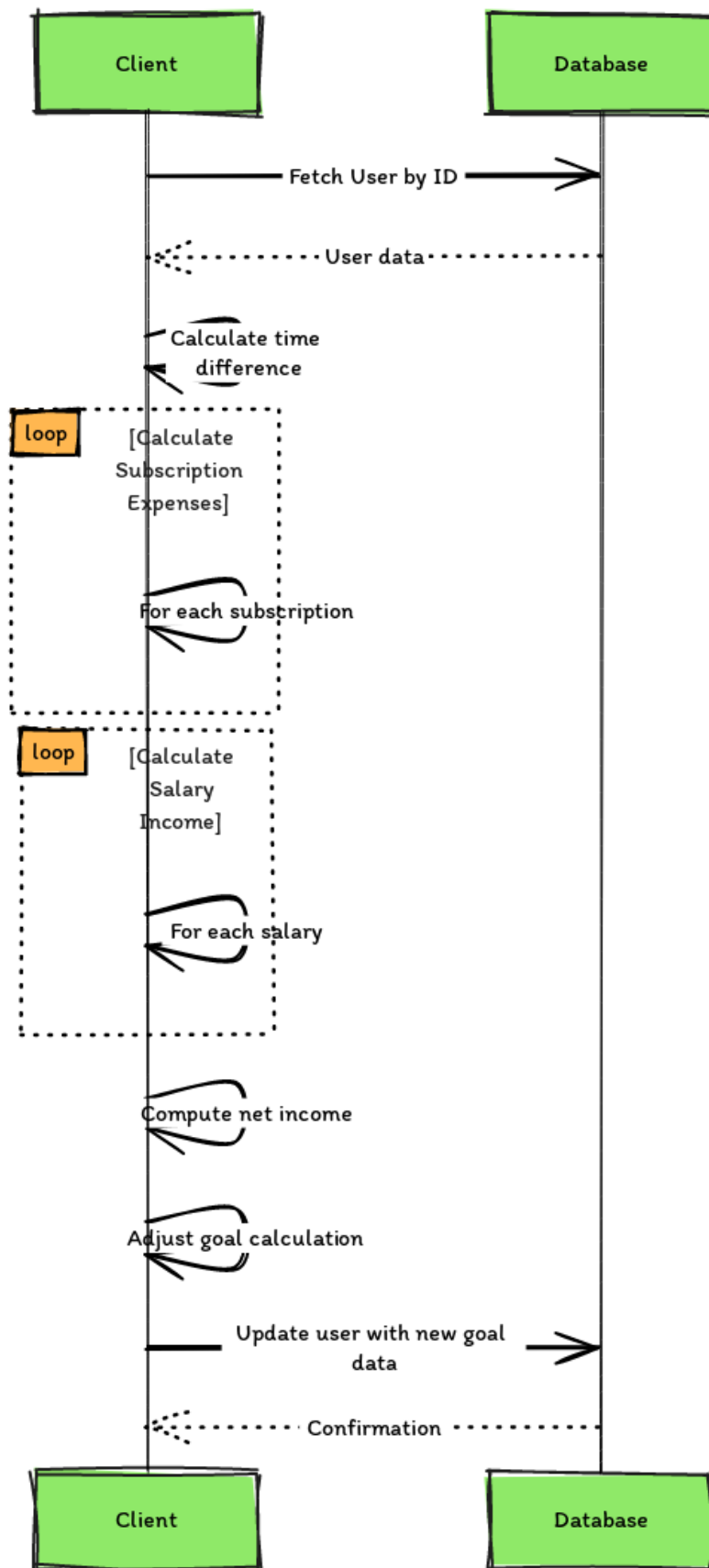
- Amount:
- Date:
- Category:
- Description (optional):
- Percentage of income to goal (optional):
- ADD**

New field optional called Percentage of income to goal is added.

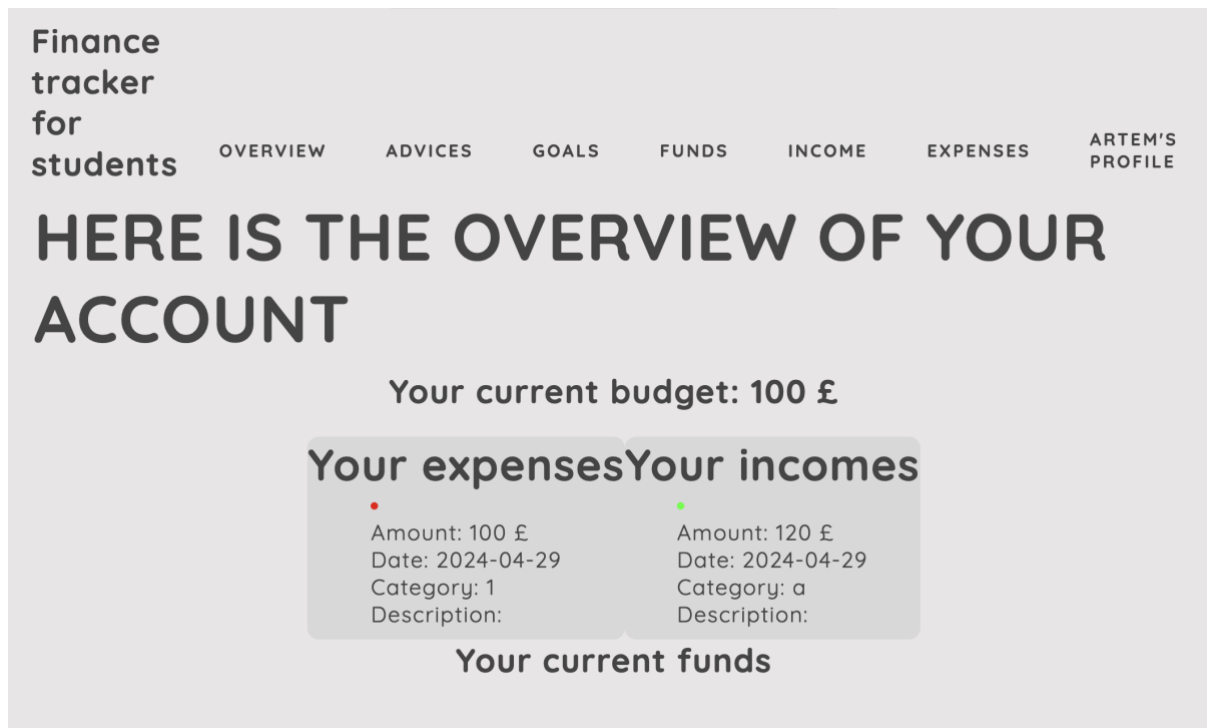
The percentage inserted by user will subtracted from 1-time income and added to the goal.

## Goal advice calculation function

Visualisation



## Overview page



This page has sorted incomes and expenses using MergeSort  
Also this page shows all funds user currently has

## Database

As my database I chose MongoDB as it is NoSQL document-oriented database, which means you store data in flexible, JSON-like documents.

User schema (class) has:

Email – contains email of user

Password – contains hashed password of user

Name – name of user

Budget – current budget of user

Updated – needed variable for track of subs and salaries.

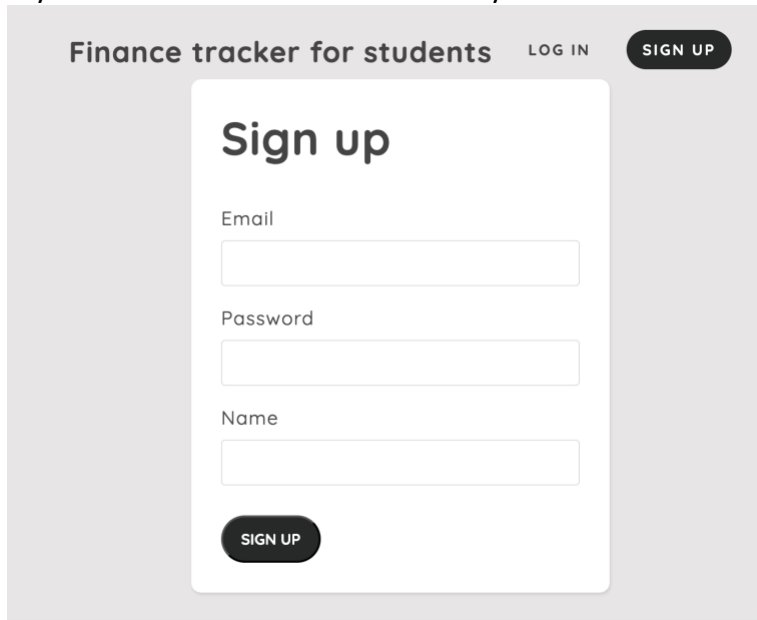
ShowGoal – for displaying of not of goal at the top of the page as on page 11

Arrays goals, funds, salaries, subs, expenses, incomes – to contain documents of users financial transactions.

When saving, each document including user itself gets a specific Id in order for easier search and use of data

## Security

My website has user authentication system



The image shows a sign-up form for a website titled "Finance tracker for students". The form is centered on a light gray background. At the top of the form, there is a header with the title "Finance tracker for students", a "LOG IN" link, and a "SIGN UP" button. The form itself has a white background and a rounded border. It contains three input fields: "Email", "Password", and "Name". Each field has a label above it and a text input box. Below the "Name" field is a "SIGN UP" button. The overall design is clean and modern.

User sign up using unique email and password which is 6 or more letters. When this password is saved into user object it is hashed using bcrypt. As soon as user signups or logins, cookie with hashed user's id is created and it is checked every GET request. As well as that it gives an ability to find out exactly which user is currently sending a request

View if you are not signed in:



The image shows the home page of the "Finance tracker for students" website. The page has a light gray background. At the top, there is a header with the title "Finance tracker for students", a "LOG IN" link, and a "SIGN UP" button. Below the header, there is a large heading "Your Guide to Personal Financial Management". Underneath this heading, there is a paragraph of text: "Welcome to your ultimate tool for tracking and improving your financial health. Our finance tracker is designed to help you monitor your spending, save more money, and achieve your financial goals. This page provides essential financial advice and tips to get you started."

View if you are signed in:

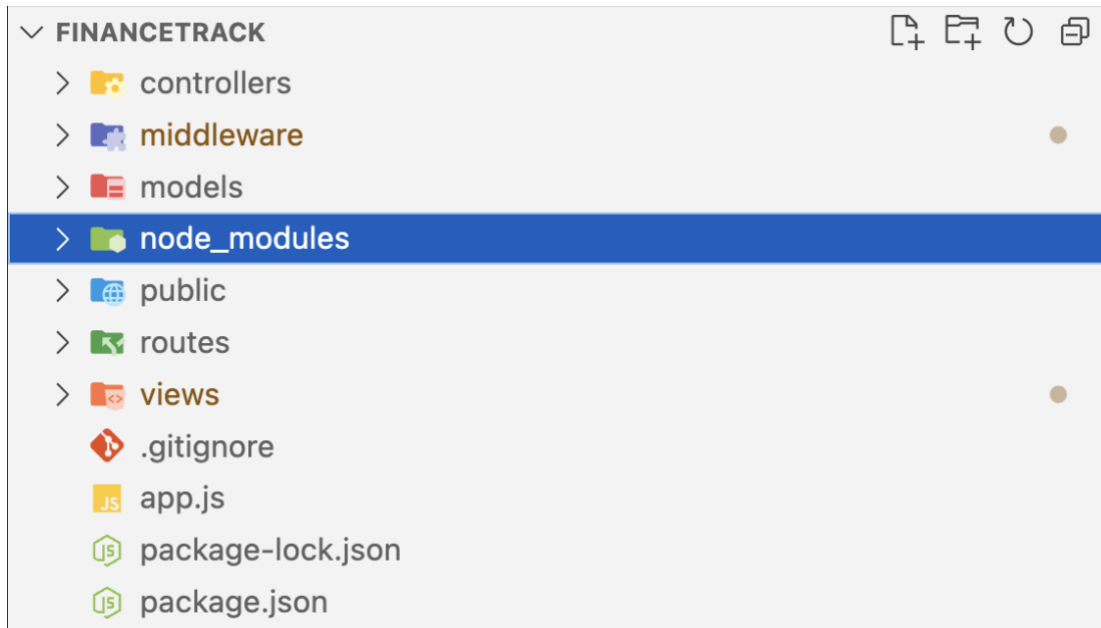


The image shows the dashboard of the "Finance tracker for students" website. The page has a light gray background. At the top, there is a header with the title "Finance tracker for students" and a navigation menu with links: "OVERVIEW", "ADVISES", "GOALS", "FUNDS", "INCOME", "EXPENSES", and "ARTEM'S PROFILE". Below the header, there is a large heading "TOOLS AND ABILITIES OF THIS TRACKER".

This is implemented using EJS

# Technical solution

## General client-server structure



## App.js

app.js is the main file which sets up the application, connects to the database and defines the routes for the application.

Here is the code:

```
const express = require('express'); //importing express
const mongoose = require('mongoose'); //importing mongoose

const authRoutes = require('./routes/authRoutes'); //importing the authRoutes
const expincRoutes = require('./routes/expincRoutes'); //importing the expincRoutes
const fundsRoutes = require('./routes/fundsRoutes'); // importing the fundsRoutes
const goalsRoutes = require('./routes/goalsRoutes'); // importing the goalsRoutes

const cookieParser = require('cookie-parser'); //importing cookie-parser
const { checkUser, changeUpdated } = require('./middleware/authMiddleware'); //importing the checkUser
middleware

const app = express(); //creating an instance of express
```



```

// middleware
app.use(express.urlencoded({ extended: true })); //for parsing incoming request bodies under the req.body
(reading data from forms)
app.use(express.static('public')); //for serving static files like css or images
app.use(express.json()); //parsing JSON data sent by the client
app.use(cookieParser()); //allows to read cookies sent back from the client

app.set('view engine', 'ejs'); //setting the view engine to ejs

// database connection

const dbURI = 'mongodb+srv://coxych:v10041997v@cluster0.l4vn38v.mongodb.net/node-auth';
mongoose.connect(dbURI, { useNewUrlParser: true, useUnifiedTopology: true, useCreateIndex:true })
  .then((result) => app.listen(3000))
  .catch((err) => console.log(err));

// routes
app.get('/', changeUpdated);
app.get('*', checkUser); //every get request will run this checkUser middleware
app.get('/', (req, res) => res.render('home'));
app.use(goalsRoutes);
app.use(expincRoutes);
app.use(authRoutes);
app.use(fundsRoutes);

```

## Models folder (database)

Models folder contains user.js file which defines UserSchema for MongoDB, hashes user password when signing up, and authorize user when logging in.

Here is the code:

```

const mongoose = require('mongoose');
const { isEmail } = require('validator');
const bcrypt = require('bcrypt');

// Create a schema for user
const userSchema = new mongoose.Schema({
  // email which should be unique and required

```

```

email: {
  type: String,
  required: [true, 'Please enter an email'],
  unique: true,
  lowercase: true,
  // here it validates that the email enter has right format
  validate: [isEmail, 'Please enter a valid email']
},
// password which should more that 6 characters and is required
password: {
  type: String,
  required: [true, 'Please enter a password'],
  minlength: [6, 'Minimum password length is 6 characters'],
},
name: {
  type: String,
},
budget:{
  type: Number,
},
updated: {
  type: Boolean,
  default: false,
},
ShowGoal: {
  type: Boolean,
  default: false,
},
// arrays for storing documents about transactions goals and funds
goals: [],
funds: [],
salaries: [],
subs: [],
expenses: [],
incomes: [],
});

// fire a function before doc saved to db
userSchema.pre('save', async function(next) {
  if (this.isNew) { // Check if the document is new

```

```

const salt = await bcrypt.genSalt(); //generate a salt
this.password = await bcrypt.hash(this.password, salt); //password hashing
this.budget = 0;
this.budget.require = [true, 'Please enter a budget'];
}
next();
});

// static method to login user
userSchema.statics.login = async function(email, password) {
  const user = await this.findOne({ email }); //find the user with the email
  if (user) {
    const auth = await bcrypt.compare(password, user.password); // compares passwords
    if (auth) {
      return user; //returns if auth is successful
    }
    throw Error('incorrect password'); //if the password is incorrect
  }
  throw Error('incorrect email'); // incorret email if the email is not found
};

const User = mongoose.model('user', userSchema); //creates a model from the schema

module.exports = User; //exports file to be used in other files

```

## Security

### Sign up

*HTML page for sign up*

```

<%- include('partials/header'); -%> //header with nav bar

<form>
  <h2>Sign up</h2>
  <label for="email">Email</label>
  <input type="text" name="email" required />
  <div class="email error"></div>
  <label for="password">Password</label>

```

```

<input type="password" name="password" required />
<div class="password error"></div>
<label for="name">Name</label>
<input type="name" name="name" required />
<div class="name error"></div>
<button class="btn">Sign up</button>
</form>
<script>
  const form = document.querySelector('form'); //grab the form
  const emailError = document.querySelector('.email.error');
  const passwordError = document.querySelector('.password.error');
  form.addEventListener('submit', async (e) => {
    e.preventDefault();
    //reset errors
    emailError.textContent = "";
    passwordError.textContent = "";

    // get values
    const email = form.email.value;
    const password = form.password.value;
    const name = form.name.value;

    try { // try to send the data to the server using post request
      const res = await fetch('/signup', {
        method: 'POST',
        body: JSON.stringify({ email, password, name }), //send the data as a json object
        headers: {'Content-Type': 'application/json'} //specify the type of data
      });
      const data = await res.json(); //get the response from the server
      if (data.errors){
        emailError.textContent = data.errors.email; //display the error message
        passwordError.textContent = data.errors.password; //display the error message
      }
      if (data.user) {
        location.assign('/') //redirect to the home page
      }
    }
    catch (err) {
      console.log(err); //log the error
    }
  }

```

```

});
</script>
<%- include('partials/footer'); -%>

```

Data is grabbed from the form user fills in and is sent to the server via post request where it is being processed.

Then it awaits for data to be returned and if the data is errors it displays them and if the data is new instance of user, then user is assigned to the home page already logged in.

### *Server side for sign up*

Password hashing is done at the moment of instantiation of the user

.pre('save') means pre saving fire following function:

```

// fire a function before doc saved to db
userSchema.pre('save', async function(next) {
  if (this.isNew) { // Check if the document is new
    const salt = await bcrypt.genSalt(); //generate a salt
    this.password = await bcrypt.hash(this.password, salt); //password hashing
    this.budget = 0;
    this.budget.require = [true, 'Please enter a budget'];
  }
  next();
});

```

Salt is secret string for hashing the password

```

const User = require('../models/user');
const jwt = require('jsonwebtoken');

// Utility for handling errors
const handleError = (err) => {
  let errors = { email: "", password: "" };

  // Specific error messages based on the error encountered
  if (err.message.includes('incorrect email')) errors.email = 'Email not registered';
  if (err.message.includes('incorrect password')) errors.password = 'Incorrect password';
  if (err.code === 11000) errors.email = 'Email already registered';
  if (err.message.includes('user validation failed')) {
    // Parse and assign validation errors
    Object.values(err.errors).forEach(({ properties }) => {

```

```

    errors[properties.path] = properties.message;
  });
}

return errors;
};

// Utility for creating JWT tokens
// This token will be used in cookies to authenticate users
const maxAge = 3 * 24 * 60 * 60; // 3 days in seconds
// secrethashing is the secret key used to hash the user ID
const createToken = (id) => jwt.sign({ id }, 'secrethashing', { expiresIn: maxAge });

// Handle user signup
const signup_post = async (req, res) => {
  try {
    const user = await User.create(req.body); // Create a new user using the User model
    const token = createToken(user._id); // Create a JWT token and hash the user ID
    res.cookie('jwt', token, { httpOnly: true, maxAge: maxAge * 1000 }); // Set the JWT token in a cookie
    res.status(201).json({ user: user._id }); // Respond with the user ID in JSON format (the res.user)
  } catch (err) {
    res.status(400).json({ errors: handleErrors(err) }); // Handle any errors using the utility function
  }
};

```

## Log in

### HTML page for log in

```
<%- include('partials/header'); -%>
```

```
<form>
```

```
  <h2>Login</h2>
```

```
  <label for="email">Email</label>
```

```
  <input type="text" name="email" required />
```

```
  <div class="email error"></div>
```

```
  <label for="password">Password</label>
```

```
  <input type="password" name="password" required />
```

```
  <div class="password error"></div>
```

```
  <button class="btn">Log in</button>
```

```
</form>
```

```
<script>
```

```
  const form = document.querySelector('form'); //grab the form
```

```
  const emailError = document.querySelector('.email.error'); //grab the error
```

```
  const passwordError = document.querySelector('.password.error'); //grab the error
```

```
  form.addEventListener('submit', async (e) => { //listen for the submit event
```

```
    e.preventDefault(); //prevent the default action
```

```
    //reset errors
```

```
    emailError.textContent = "";
```

```
    passwordError.textContent = "";
```

```
    // get values
```

```
    const email = form.email.value; //get the email value
```

```
    const password = form.password.value; //get the password value
```

```
    try {
```

```
      const res = await fetch('/login', { //send the data to the server using post request
```

```
        method: 'POST',
```

```
        body: JSON.stringify({ email, password }),
```

```
        headers: {'Content-Type': 'application/json'}
```

```
      });
```

```
      const data = await res.json(); //get the response from the server
```

```
      console.log(data);
```

```
      if (data.errors){
```

```
        emailError.textContent = data.errors.email;
```

```
        passwordError.textContent = data.errors.password;
```

```
      }
```

```

    if (data.user) {
      location.assign("/") //redirect to the home page if the res is user
    }
  }
  catch (err) {
    console.log(err); //log the error
  }
});
</script>

```

```

<%- include('partials/footer'); -%>

```

Log in page also works via post request sending

*Server side for log in*

// Handle user login

```

const login_post = async (req, res) => {
  try {
    const user = await User.login(req.body.email, req.body.password); // Log the user in using the User model
    const token = createToken(user._id); // Create a JWT token and hash the user ID
    res.cookie('jwt', token, { httpOnly: true, maxAge: maxAge * 1000 }); // Set the JWT token in a cookie
    res.status(200).json({ user: user._id }); // Respond with the user ID in JSON format (the res.user)
  } catch (err) {
    res.status(400).json({ errors: handleErrors(err) }); // Handle any errors using the utility function
  }
};

```

User.login function was shown previously with model code but I will repeat it:

// static method to login user

```

userSchema.statics.login = async function(email, password) {
  const user = await this.findOne({ email }); //find the user with the email
  if (user) {
    const auth = await bcrypt.compare(password, user.password); // compares passwords
    if (auth) {
      return user; //returns if auth is successful
    }
    throw Error("incorrect password"); //if the password is incorrect
  }
  throw Error("incorrect email"); // incorret email if the email is not found
};

```



## UserCheck

```
const checkUser = (req, res, next) => {
  const token = req.cookies.jwt; //get the token from the cookie
  if (token) { //if the token exists
    jwt.verify(token, 'secrethashing', async (err, decodedToken) => { //verify the token
      if (err) {
        res.locals.user = null; //if there is an error, set the user to null
        next();
      } else {
        // if the token is verified, find the user with the id in the token
        let user = await User.findById(decodedToken.id);
        res.locals.user = user; //set the user to the user found
        //now user can be using in html via EJS
        await regularCheck(decodedToken.id); //check if the user has unupdated subs or incomes
        next();
      }
    });
  } else {
    res.locals.user = null; //if there is no token, set the user to null
    next();
  }
};
```

CheckUser happens every get request, here is the code from app.js where this is assigned:

```
app.get('*', checkUser); //every get request will run the checkUser middleware
```

This is extremely useful to hide some information from not registered users like shown in Security Design or you can use data contained within user object to display specific things like name, this is done using EJS.

Here is the code of header/navbar HTML with EJS in it:

```

<nav>
  <h1><a href="/">Finance tracker for students</a></h1>
  <ul>
    <% if (user) { %> //if the user is logged in functionality is revealed
    <li><a href="/overview">Overview</a></li>
    <li><a href="/advices">Advices</a></li>
    <li><a href="/goals">Goals</a></li>
    <li><a href="/funds">Funds</a></li>
    <li><a href="/income">Income</a></li>
    <li><a href="/expenses">Expenses</a></li>
    // here the user.name (name of the user) is passed to HTML
    <li><a href="/profile"><%= user.name %>'s Profile</a></li>
    <% } else { %> //if the user is not logged in the functionality is hidden
    <li><a href="/login">Log in</a></li>
    <li><a href="/signup" class="btn">Sign up</a></li>
    <% } %>
  </ul>
</nav>

```

EJS JavaScript is separated from the using `<%%>` and if data is need to be passed to html `<%= %>` is used.

## 1-time incomes and expenses

I decided to combine technical solution for 1-tume incomes and expenses into 1 chapter as they are very similar with only a few differences. AddIncome function is more complex than AddExpense because of process of adding money to goal I have shown in design. More on this in goal section.

### HTML for income expense page

```

<%- include('partials/header'); -%>

<header>
  <h2>HERE YOU CAN MANAGE YOUR EXPENSES</h2>
</header>

```

```
<h2>Your current budget = <%= user.budget %> £</h2> // display budget using EJS
```

```
<h3>You can add your expenses here</h3>
```

```
<div class="row-layout">
```

```
<br>
```

```
<form action="/expenses" method="post">
```

```
<label for="amount">Amount</label>
```

```
<input type="number" step="0.01" name="amount" required>
```

```
<label for="date">Date</label>
```

//here we use the new Date().toISOString().split('T')[0] to get the current date in the format that the input type date requires

```
<input type="date" name="date" value="<%= new Date().toISOString().split('T')[0] %>" required>
```

```
<label for="category">Category</label>
```

```
<input type="text" name="category" required>
```

```
<label for="description">Description (optional)</label>
```

```
<input type="text" name="description" >
```

```
<button class="btn" type="submit">Add</button>
```

```
</form>
```

```
<script>
```

```
const form = document.querySelector('form'); //grab the form
```

```
form.addEventListener('submit', async (e) => {
```

```
  e.preventDefault();
```

```
  const amount = form.amount.value; //get the amount value
```

```
  const date = form.date.value; //get the date value
```

```
  const category = form.category.value; //get the category
```

```
  const description = form.description.value; //get the description
```

```
  try {
```

```
    const res = await fetch('/expenses', { //sending post req to server
```

```
      method: 'POST',
```

```
      body: JSON.stringify({ amount, date, category, description }),
```

```
      headers: { 'Content-Type': 'application/json' }
```

```
    });
```

```
    const data = await res.json();
```

```
    if (data) { // if there is a response refresh expenses
```

```
      location.assign('/expenses');
```

```
    }
```

```
  } catch (err) {
```

```
    console.log(err.message);
```

```
  }
```

```
});
```

```

</script>
<div class="scroll-r">
  <h2>Your 6 latest expenses:</h2>
  <% for (let i = 0; i < 6 && i < user.expenses.length; i++) { %> //for loop to show last 6 expnses
    <li>
      // sidplaying objects from expenses array
      <p>Amount: <%= user.expenses[user.expenses.length - 1 - i].amount %> £</p>
      <p>Date: <%= user.expenses[user.expenses.length - 1 - i].date %></p>
      <p>Category: <%= user.expenses[user.expenses.length - 1 - i].category %></p>
      <% if (user.expenses[user.expenses.length - 1 - i].description){ %>
        <p>Description: <%= user.expenses[user.expenses.length - 1 - i].description %></p>
      <% } else { %>
        <p></p>
      <% } %>
    </li>
  <% } %>
</div>
</div>

```

## Server side of exp/inc

```

module.exports.expenses_post = ('/expenses', async (req, res) => {
  // get data from request
  const { date, category, amount, description } = req.body;
  try {
    const id = getUserId(req); //this function gets id of user who sends the request
    const newExpense = { date, category, amount, description };
    // as here id is a promise .then() is used to wait for it to resolve
    id.then(async (id) => {
      await addExpense(id, newExpense); //function which pushes expense to array for user with this id
      res.status(200).json({message: 'Expense added'}); //res sending
    });
  } catch (err) {
    console.log(err);
  }
});

```

*getUserId function:*

```
const getUserId = (req) =>{
  const token = req.cookies.jwt; //token is taken from the cookie attached to the request
  if (token){ // if there JWT in the cookie
    return new Promise((resolve, reject) => { //promise as this is async
      jwt.verify(token, 'secret hashing', (err, decodedToken) =>{ //decodes token and checks if it is valid
        // valid means signed with correct secret word and not expired
        if (err){
          console.log(err.message);
          reject(err); //reject promise
        }
        else{
          resolve(decodedToken.id); //if verification is successful then id from JWT (user's id) is returned
        }
      })
    });
  }
  else{
    return null; // if no JWT is found in the cookies returns null
  }
}
```

*AddExpense function*

```
async function addExpense(userId, expense) {
  try {
    const user = await User.findById(userId); //finds user by id, findById is prebuilt function in MongoDB
    const update = { //update is an object that has all new data that should be updated
      budget: user.budget - parseFloat(expense.amount), //expense subtraction from budget
      $push: {
        expenses: expense, // Push the expense object to the expenses array
      },
    };
    await user.updateOne(update); //wait till new data is added and updated to user, updateOne is prebuilt
  } catch (err) {
    console.log(err.message);
  }
}
```

## Subs and salaries addition and tracking

Again, I will combine subs and constant incomes together as they are similar in structure.

## HTML subs and constant incomes

```
<h2>ADD YOUR SUBSCRIPTIONS OR USUAL EXPENSES HERE</h2>
<br>
<div class="row-layout">
<form action="/addsubs" method="post">
  <label for="amount">Amount</label>
  <input type="number" step="0.01" name="amount" required>
  <label for="howoften">How often</label>
  <select name="howoften" id="howoften"> //select bar which allows to choose frequency of payment
    <option value="daily">Daily</option>
    <option value="weekly">Weekly</option>
    <option value="monthly">Monthly</option>
  </select>
  <label for="date">Next date of payment</label>
  //again conversion of current data into the required format
  <input type="date" name="date" value="<%= new Date().toISOString().split('T')[0] %>" required>
  <label for="category">Category</label>
  <input type="text" name="category" required>
  <label for="description">Description (optional)</label>
  <input type="text" name="description" >
  <button class="btn" type="submit">Add</button>
</form>
<script>
const form2 = document.querySelector('form');
form2.addEventListener('submit', async (e) => { //adds submit button listener to form
  e.preventDefault(); //default action prevention as it is reloading the page
  //getting values from form
  const amount = form2.amount.value;
  const date = form2.date.value;
```

```

const howoften = form2.howoften.value;
const category = form2.category.value;
const description = form2.description.value;
try {
  const res = await fetch('/addsubs', { //post req sending to server
    method: 'POST',
    body: JSON.stringify({ amount, date, category, description, }),
    headers: { 'Content-Type': 'application/json' }
  });
} catch (err) {
  console.log(err.message);
}
});
</script>
<div class="scroll-r">
  <h2>All you subscriptions and constant expenses</h2>
  <% user.subs.reverse().forEach((sub) => { %> //reverse as subs were in wrong order when displayed
  <div class="row-layout">
    <li>
      //displaying of data from sub object in subs array
      <p>Amount: <%= sub.amount %> £</p>
      <p>Next date: <%= sub.date %></p>
      <p>Category: <%= sub.category %></p>
      <p>How Often: <%= sub.howoften %></p>
      <% if (sub.description){ %>
        <p>Description: <%= sub.description %></p>
      <% } else { %>
        <p></p>
      <% } %>
      //delete button which contains id and sub id in itself for further delete req
      <button class="btn delete-btn" data-id="<%= sub.id %>" data-usid="<%= user._id %>">DELETE</button>
    </li>
  </div>
  <% }); %>
</div>
<script>
document.addEventListener('DOMContentLoaded', () => {
  document.querySelectorAll('.delete-btn').forEach(button => { //for each as there might be several subs
    button.addEventListener('click', function() { //listener to delet button
      //extraction of data from delete button (user id and sub id)

```

```

const subId = this.getAttribute('data-id');
const userId = this.getAttribute('data-usid');
try {
  fetch(`/user/${userId}/sub/${subId}`, { //delete req sending
    //this req contains data in itself as it is delete req
    method: 'DELETE',
    headers: {
      'Content-Type': 'application/json',
    }
  }).then(() => {
    location.reload(); //reload of bage after req is sent
  });
} catch {
  console.log('failed to delete sub');
}
});
});
});
</script>
</div>
<%- include('partials/footer'); -%>

```

## Server side of subs/salaries addition

### Post req handling

```

module.exports.addsubs_post = (req, res) => {
  //get data from the req
  const { date, category, howoften, amount, description } = req.body;
  try {
    const id = getUserId(req); //get user id (same function as previously)
    const newSub = { date, category, amount, description, howoften }; // crete newsub object
    id.then(async (id) => {
      await addSubs(id, newSub) //add sub to subs array
      res.status(200).redirect('/expenses');
    });
  } catch (err) {
  }
}

```



```

        console.log(err);
    }
});

addSubs function
async function addSalary(userId, salary) {
    try{
        const user = await User.findById(userId);
        const salaryWithId = {
            ...salary,
            id: uuidv4(), // Add a unique ID to the salary object
        };
        const update = {
            updated: true, //to show that new sub or salarie is added and its date should be checked
            $push: {
                salaries: salaryWithId,
            },
        };
        await user.updateOne(update);
    }catch (err) {
        console.log(err.message);
    }
}

```

In case with subs and salaries a unique id is a necessity as later you need to search for a particular one in order to update it

## Delete req

```

module.exports.deletesub_delete = async (req, res) => {
    const { userId, subId } = req.params;
    try {
        // Find the user by their ID
        const user = await User.findById(userId);
        //remove the sub from the user's subs array
        user.subs = user.subs.filter(sub => sub.id !== subId);
        // Save the user document after modification
        await user.save();

        // Send a success response back to the client
        res.status(200).json({ message: 'Sub deleted successfully' });
    }
}

```

```

    } catch (error) {
      // Handle any errors that occur during the process
      res.status(500).json({ message: 'Error deleting Sub', error: error.message });
    }
  }
}

```

## Tracking of subs and salaries

### *RegularCheck function*

Track is implemented using regularCheck function which is run when CheckUser function is run as shown in Security part of Technical Solution (it is run every get request). This function firstly checks subs and salaries arrays and checks if there are any subs or salaries that need to be updated, if there are, it updates them using checkAndAddIncomes/Expenses

```

async function regularCheck(userId) {
  try {
    const user = await User.findById(userId); //finds user by id
    for (let sub of user.subs) {
      let subDate = new Date(sub.date);
      if (subDate <= now) {
        user.updated = true;
        break;
      }
    }
    for (let salary of user.salaries) {
      let salaryDate = new Date(salary.date);
      if (salaryDate <= now) {
        user.updated = true;
        break;
      }
    }
    if (user.updated === true) {
      await checkAndAddIncomes(userId);
      await checkAndAddExpenses(userId);
      user.updated = false; //returns value of updated
    }
  } catch (err) {
    console.log(err.message);
  }
}

```

### *checkAndAdd functions*

checkAndAdd functions are similar so I'll provide code only for expenses:

```
async function checkAndAddExpenses(userId) {
  const now = new Date(); // declare the current date
  let user = await User.findById(userId); // Fetch the user and their subscriptions
  for (let sub of user.subs) { // for all subs
    let subDate = new Date(sub.date); //get date of sub
    if (subDate <= now) {
      await addExpense(userId, { //adding expense which is has sub data like amount, date, category
        date: sub.date,
        amount: sub.amount,
        description: sub.description,
        category: sub.category
      });
      // Correctly update subDate based on frequency
      switch (sub.howoften) {
        case 'daily':
          subDate.setDate(now.getDate() + 1);
          break;
        case 'weekly':
          subDate.setDate(now.getDate() + 7);
          break;
        case 'monthly':
          subDate.setMonth(now.getMonth() + 1);
          break;
      }
      user.updatedAt = new Date();
      // Format the date as a string in "YYYY-MM-DD" format before reassigning
      // as it will be given in this format: 2021-09-01T00:00:00.000Z
      const formattedDate = subDate.toISOString().split('T')[0]; //splitting the date and time and taking only date
      sub.date = formattedDate; //reassigning the date
    }
  }
  user.markModified('subs') // Mark the subs property as modified
  await user.save(); // Save the user document
}
```

# Funds page

## HTML funds

```
<%- include('partials/header'); -%>
<header>
  <h2>HERE YOU CAN MANAGE YOUR FUNDS</h2>
</header>

<h2>You can set goals and create funds here</h2>
<br>
<h2>Your current budget: <%= user.budget %> £</h2>

<h2>Create a fund here:</h2>
<div class="row-layout">
  <br>
  <form action="/addfunds" method="post">
    <label for="amount">Amount to raise</label>
    <input type="number" step="0.01" name="amount" required>
    <label for="name">Name</label>
    <input type="text" name="name" required>
    <button class="btn" type="submit">Add</button>
  </form>
  <script>
    const form = document.querySelector('form');
    form.addEventListener('submit', async (e) => {
      e.preventDefault();
      //get values from the form
      const amount = form.amount.value;
      const name = form.name.value;
      try {
        //send post req
        const res = await fetch('/addfunds', {
          method: 'POST',
          body: JSON.stringify({ amount, name }),
          headers: { 'Content-Type': 'application/json' }
        });
        const data = await res.json();
```

```

    if (data) {
        location.assign('/funds');
    }
} catch (err) {
    console.log(err.message);
}
});
</script>

<div>
    <h2>Your current funds</h2>
    <% user.funds.reverse().forEach((fund) => { %>
    <li>
        <!-- Representation of funds on the right of the page-->
        <p>To raise: <%= fund.amount %> £</p>
        <div class="fundBarContainer" data-name="<%= fund.name %>" data-amount="<%= fund.amount %>" data-
raised="<%= fund.raisedAmount %>">
            <!-- fundBar is the green filling part of funds-->
            <div class="fundBar" style="width: 0%;"></div>
            <p> <%= fund.name %> and raised: <%= fund.raisedAmount%></p>
            <!-- if some mone is added to fund excess button instead of delete appears-->
            <p class="excess"></p>
            <% if (fund.raisedAmount > 0){ %>
                <button class="btn excess-btn" data-usid="<%= user._id %>" data-id="<%= fund.id %>" display=""
>Withdraw fund</button>
            <% } else { %>
                <button class="btn delete-btn" data-usid="<%= user._id %>" data-id="<%= fund.id %>">DELETE</button>
            <% } %>
        </div>
    </script>
    document.addEventListener('DOMContentLoaded', () => {
        document.querySelectorAll('.fundBarContainer').forEach(container => {
            const toRaise = parseInt(container.getAttribute('data-amount'), 10);
            const raised = parseInt(container.getAttribute('data-raised'), 10);
            // "?" states if there is some money then next operation should be performed
            const percentageRaised = raised > 0 ? (raised / toRaise) * 100 : 0;
            if (percentageRaised >= 100) { // this is needed as CSS can excess 100% and go out of boundaries of
container
                container.querySelector('.fundBar').style.width = '100%';
            }
        });
    });

```

```

        container.querySelector('.excess').textContent = `You have raised more than the goal, excess: ${raised
- toRaise} £`;
    } else {
        container.querySelector('.fundBar').style.width = `${percentageRaised}%`;
    }
    //excess btn is for withdrawing money
    container.querySelector('.excess-btn').style.display = 'block';
    document.querySelectorAll('.excess-btn').forEach(button => {
        button.addEventListener('click', async function() {
            const fundId = this.getAttribute('data-id');
            const userId = this.getAttribute('data-usid');
            try { //post req is sent withdraw
                await fetch(`/user/${userId}/fund/${fundId}/withdraw`, {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json',
                    }
                }).then(() => {
                    location.reload();
                });
            } catch {
                console.log('failed to delete withdraw');
            }
        });
    });
});

document.querySelectorAll('.delete-btn').forEach(button => {
    button.addEventListener('click', function() {
        const fundId = this.getAttribute('data-id');
        const userId = this.getAttribute('data-usid');
        try { //delete req
            fetch(`/user/${userId}/fund/${fundId}`, {
                method: 'DELETE',
                headers: {
                    'Content-Type': 'application/json',
                }
            }).then(() => {
                location.reload();
            });
        });
    });
});

```

```

        } catch {
            console.log('failed to delete fund');
        }
    });
});
});
</script>
</li>
<% }) %>
</div>
</div>
<br>
<br>
<div>
    <h2>You can add or withdraw money from your funds here</h2>
    <p>The amount would be added or taken from the budget</p>
    <br>
    <form action="/updatefunds" method="post">
        <label for="fund">Choose a fund to update</label>
        <!-- gives you list of all existing funds-->
        <select name="fundId" id="fundId">
            <% user.funds.reverse().forEach((fund) => { %>
                <option value="<%= fund.id %>"><%= fund.name %></option>
            <% }) %>
        </select>
        <label for="type">Choose a type</label>
        <!-- either withdraw money or add on fund-->
        <select name="type" id="type">
            <option value="add">Add</option>
            <option value="withdraw">Withdraw</option>
        </select>
        <label for="amount">Amount to add</label>
        <input type="number" step="0.01" name="amount" required>
        <button class="btn" type="submit">Add</button>
    </form>
</script>
const form2 = document.querySelector('form');
form2.addEventListener('submit', async (e) => {
    e.preventDefault();
    const fundId = form2.fundId.value;

```

```

const type = form2.type.value;
const amount = form2.amount.value;
try { //post req to update funds
  const res = await fetch('/updatefunds', {
    method: 'POST',
    body: JSON.stringify({ fundId, type, amount }),
    headers: { 'Content-Type': 'application/json' }
  });
  location.assign('/funds');
} catch (err) {
  console.log(err.message);
}
});
</script>
</div>

<%- include('partials/footer'); -%>

```

## addFunds request

```

module.exports.addfunds_post = ('/addfunds', async (req, res) => {
  const { name, amount } = req.body;
  try {
    const id = getUserId(req);
    const newFund = { name, amount };
    id.then(async (id) => {
      await addFunds(id, newFund)
      res.status(200).json({ message: 'Funds added successfully' });
    });
  } catch (err) {
    console.log(err);
  }
}

```



```
});
```

*addFunds function*

```
async function addFunds(userId, fund) {  
  try{  
    const user = await User.findById(userId);  
    const fundWithId = {  
      ...fund,  
      id: uuidv4(), // here a unique ID is also usefull  
      raisedAmount: 0,  
    };  
    const update = {  
      updated: true,  
      $push: {  
        funds: fundWithId,  
      },  
    };  
    await user.updateOne(update);  
  } catch (err) {  
    console.log(err.message);  
  }  
}
```

## Withdraw post req

```
module.exports.withdrawfund_withdraw = ('/user/:userId/fund/:fundId/withdraw', async (req, res) => {  
  const { userId, fundId } = req.params; //extracting data from the req parameters  
  try{  
    await withdrawFund(userId, fundId)
```

```

    res.status(200).redirect('/funds');
  } catch (err) {
    console.log(err);
  }
});

```

*withdrawFund function*

```

async function withdrawFund(userId, fundId) {
  try {
    const user = await User.findById(userId);
    const fund = user.funds.find(fund => fund.id === fundId);
    user.budget += fund.raisedAmount; //add the amount raised to the badget
    user.funds = user.funds.filter(fund => fund.id !== fundId); //delete from funds array
    user.markModified('funds');
    await user.save(); //save changes
  } catch (err) {
    console.log(err.message);
  }
}

```

## Delete funds req

```

module.exports.deletefund_delete = ('/user/:userId/fund/:fundId', deleteFund);
const deleteFund = async (req, res) => {
  const { userId, fundId } = req.params;
  try {
    const user = await User.findById(userId);
    user.funds = user.funds.filter(fund => fund.id !== fundId); //delete fund from the array
    user.markModified('funds');
    await user.save();
    res.status(200).json({ message: 'Fund deleted successfully' });
  } catch (err) {
    console.log(err);
  }
}

```

## Updatefunds req

```

module.exports.updatefunds_post = ('/updatefunds', async (req, res) => {
  const { type, amount, fundId } = req.body;
  try {
    const id = getUserId(req); //get id from req
    id.then(async (id) => {
      const user = await User.findById(id);
      const fund = user.funds.find(fund => fund.id === fundId); //finds the fund by the unique id created
      previously
      if (fund) {
        if (type === 'add') {
          //addition from budget
          fund.raisedAmount += parseFloat(amount);
          user.budget -= parseFloat(amount);
        } else {
          //putting money from fund on budget
          fund.raisedAmount -= parseFloat(amount);
          user.budget += parseFloat(amount);
        }
        user.markModified('funds');
        await user.save(); //save changes
        res.status(200).redirect('/funds');
      } else {
        res.status(404).json({ message: 'Fund not found' });
      }
    });
  } catch (err) {
    console.log(err);
  }
});

```

# Goal

## HTML for goal

Goal can only be added if user has at least 1 sub and 1 salary. Also, goal can be updated via goal page and using income page as shown in the design section where you just insert percentage of income to add to the goal. I'll provide income page code as well.

```
<%- include('partials/header'); -%>

<header>
  <h2>HERE YOU CAN MANAGE YOUR GOALS</h2>
</header>
<div>
  <h2>How goals work?</h2>
  <p>Goal includes amount to save and the date until which you want to accomplish this goal.</p>
  <p>We will provide you with data and advices on how to reach desired</p>
</div>
<br>
<h2>Your current budget: <%= user.budget %> £</h2>
<br>
<!-- checks if user has at least 1 sub and 1 salary so advice can be calculated-->
<% if (user.subs.length > 0 && user.salaries.length > 0 && user.goals.length == 0) { %>
<h2>Add your first goal</h2>
<br>
<form action="/addgoals" method="post">
  <label for="amount">Amount to save</label>
  <input type="number" name="amount" step="0.01" required>
  <label for="date">Date until which to save</label>
  <input type="date" name="date" value="<%= new Date().toISOString().split('T')[0] %>" required>
  <button class="btn" type="submit">Add</button>
</form>
<script>
  const form = document.querySelector('form');
  form.addEventListener('submit', async (e) => {
    e.preventDefault();
```

```

const date = form.date.value;
const amount = form.amount.value;
try { //addgoals post req
  const res = await fetch('/addgoals', {
    method: 'POST',
    body: JSON.stringify({ amount, date }),
    headers: { 'Content-Type': 'application/json' }
  });
  const data = await res.json();
  if (data) {
    location.assign('/goals');
  }
} catch (err) {
  console.log(err.message);
}
});
</script>
<% } else { %>
  <p>To use goals you should have at least 1 subscription and 1 source of constant income</p>
<% } %>

<div>
  <% user.goals.forEach((goal) => { %>
    <h2>Your current goal</h2>
    <li>
      <p>Amount to save: <%= goal.amount %> £</p>
      <p>Date until which to save: <%= goal.date %></p>
      <div class="row-layout">
        <!-- if goal is new and no money save delete button is revealed-->
        <button class="btn delete-btn"
          <% if (goal.savedAmount == 0) { %>
            style="display: block"
          <% } else { %>
            style="display: none"
          <% } %>
          data-usid="<%= user._id %>" data-id="<%= goal.id %>">
          DELETE</button>

        <!-- if goal is updated then display calculate btn-->
        <button class="btn calculate-btn"

```

```

<% if (!goal.updated) {%>
    style="display: block"
<%} else {%>
    style="display: none"
<%}%>
data-usid="<%= user._id %>" data-id="<%= goal.id %>">
CALCULATE</button>

<!-- user can either hide or show the goal at the top of the page-->
<button class="btn hide-show-btn" data-usid="<%= user._id %>">
    <% if (!user.ShowGoal) { %>
        SHOW AT THE TOP</button>
    <% } else { %>
        HIDE</button>
    <% } %>

    <!-- withdraw goal button is there if some money is saved-->
<button class="btn withdraw-btn"
<% if (goal.savedAmount > 0) { %>
    style="display: block"
<% } else { %>
    style="display: none"
<% } %>
data-usid="<%= user._id %>" data-id="<%= goal.id %>">
WITHDRAW</button>
</div>

<!-- this chunk of code displays financial advice for the user-->
<% if (goal.updated) {%>
    <p>We analysed your goal:</p>
    <% if (goal.calcValue > 0) {%>
        <p>You can easily reach your goal in saving if you won't spend more than <%= goal.calcValue %> per
day</p>
        <p>This is approximate calculation without including your expenses</p>
    <% } else {%>
        <p>To reach your goal you have to gain at least <%= goal.calcValue * -1 %> £ each day</p>
        <p>This is approximate calculation without including your expenses</p>
    <% } %>
<% } %>

<h1>Amount saved: <%= goal.savedAmount %> £</h1>
<form action="/user/{userId}/goal/{goalId}/add" method="post">
    <label for="amount">Amount to add to goal</label>
    <input type="number" name="amount" step="0.01" required>

```

```

    <button class="btn" type="submit2" data-usid="<%= user._id %>" data-id="<%= goal.id %>">Add</button>
</form>
<script>
const form2 = document.querySelector('form');
form2.addEventListener('submit', async (e) => {
    e.preventDefault();
    const amount = form2.amount.value;
    const goalId = form2.querySelector('button').getAttribute('data-id');
    const userId = form2.querySelector('button').getAttribute('data-usid');
    try { //money addition through the goal page
        const res = await fetch(`/user/${userId}/goal/${goalId}/add`, {
            method: 'POST',
            body: JSON.stringify({ amount }),
            headers: { 'Content-Type': 'application/json' }
        });
        const data = await res.json();
        if (data) {
            location.assign('/goals');
        }
    } catch (err) {
        console.log(err.message);
    }
});
</script>
</li>
<% %>
</div>
<script>
document.querySelectorAll('.delete-btn').forEach(button => {
    button.addEventListener('click', async function() {
        const goalId = this.getAttribute('data-id');
        const userId = this.getAttribute('data-usid');
        try { //deleting the goal
            await fetch(`/user/${userId}/goal/${goalId}/delete`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                }
            });
        }
    }).then(() => {
        location.reload();
    });
});

```

```

    });
  } catch {
    console.log('failed to delete goal');
  }
});
});
document.querySelector('.calculate-btn').addEventListener('click', async function() {
  const goalId = this.getAttribute('data-id');
  const userId = this.getAttribute('data-usid');
  try { //calculate button which initialise the function to calculate the advice
    await fetch(`/user/${userId}/goal/${goalId}/calculate`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      }
    }).then(() => {
      location.reload();
    });
  } catch {
    console.log('failed to calculate goal');
  }
});
document.querySelector('.withdraw-btn').addEventListener('click', async function() {
  const goalId = this.getAttribute('data-id');
  const userId = this.getAttribute('data-usid');
  try { //withdrawing goal
    await fetch(`/user/${userId}/goal/${goalId}/withdraw`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      }
    }).then(() => {
      location.reload();
    });
  } catch {
    console.log('failed to withdraw goal');
  }
});
document.querySelector('.hideshow-btn').addEventListener('click', async function() {
  const userId = this.getAttribute('data-usid');

```



```

    try { //hiding/showing goal
      await fetch(`/user/${userId}/hideshowgoal`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        }
      }).then(() => {
        location.reload();
      });
    } catch {
      console.log('failed to show goal');
    }
  })
</script>
<%- include('partials/footer'); -%>

```

## Addgoals req

```

module.exports.addgoals_post = ('/addgoals', async (req, res) => {
  const { amount, date } = req.body;
  try { //
    const id = getUserId(req);
    const newGoal = { amount, date };
    id.then(async (id) => {
      await addGoals(id, newGoal)
      res.status(200).json({ message: 'Goal added successfully' });
    });
  } catch (err) {
    console.log(err);
  }
});

```

### *AddGoals function*

```

async function addGoals(userId, goal) {
  try{
    const user = await User.findById(userId);
    const goalWithId = {
      ...goal,
      id: uuidv4(), //unique id creation
      savedAmount: 0,

```

```

    calcValue: 0,
    updated: false,
  };
  const update = {
    updated: true,
    $push: {
      goals: goalWithId,
    },
  };
  await user.updateOne(update);
} catch (err) {
  console.log(err.message);
}
}

```

## Add money to goal req

```

module.exports.addAmountToGoal = ('/addamount', async (req, res) => {
  const { userId, goalId } = req.params;
  const { amount } = req.body;
  try{
    await addAmountToGoal(userId, goalId, amount)
    res.status(200).json({ message: 'Goal updated successfully' });
  } catch(err){
    console.log(err);
  }
})

```

### *addAmountToGoal function*

```

const addAmountToGoal = async (userId, goalId, amount) => {
  try {
    const user = await User.findById(userId);
    const goal = user.goals.find(goal => goal.id === goalId);
    goal.savedAmount += parseFloat(amount);
    user.budget -= parseFloat(amount);
    user.budget = parseFloat(user.budget.toFixed(2)); // Parse to float and fix to 2 decimal places, this was
    user.markModified('goals');
    await user.save();
  }
}

```

```

    } catch (err) {
      console.log(err.message);
    }
  }
}

```

## Delete goal req

```

module.exports.deletegoal = ('user/:userId/goal/:goalId/delete', deleteGoal);
const deleteGoal = async (req, res) => {
  const { userId, goalId } = req.params;
  try {
    const user = await User.findById(userId);
    user.goals = user.goals.filter(goal => goal.id !== goalId);
    user.markModified('goals');
    await user.save();
    res.status(200).json({ message: 'goal deleted successfully' });
  } catch (err) {
    console.log(err);
  }
}

```

## Withdraw goal req

```

module.exports.withdrawgoal = ('user/:userId/goal/:goalId/withdraw', async (req, res) => {
  const { userId, goalId } = req.params;
  try{
    withdrawGoal(userId, goalId)
    res.status(200).redirect('/goals');
  }catch(err){
    console.log(err);
  }
});

```

*withdrawGoal function*

```

const withdrawGoal = async (userId, goalId) => {
  try {
    const user = await User.findById(userId);
    const goal = user.goals.find(goal => goal.id === goalId);
    user.budget += goal.savedAmount;
    user.budget = parseFloat(user.budget.toFixed(2)); // Parse to float and fix to 2 decimal places
  }
}

```

```

    user.goals = user.goals.filter(goal => goal.id !== goalId);

    user.markModified('goals');
    await user.save();
  } catch (err) {
    console.log(err.message);
  }
}

```

## Hide/Show goal req

```

module.exports.hideshowgoal = ('user/:userId/hideshowgoal', async (req, res) => {
  const { userId } = req.params;
  const user = await User.findById(userId);
  user.ShowGoal = !user.ShowGoal; //updates ShowGoal bool parameter of user
  user.markModified('ShowGoal');
  await user.save();
  res.status(200).json({ message: 'Show goals updated successfully' });
});

```

## Calculate req

```

module.exports.calculategoal = ('user/:userId/goal/:goalId/calculate', async (req, res) => {
  const { userId, goalId } = req.params;
  const user = await User.findById(userId);
  const goal = user.goals.find(goal => goal.id === goalId); //finds the goal with the id
  await goalsAlgorithm(userId, goal);
  res.status(200).json({ message: 'Goal calculated successfully' });
});

```

### *goalsAlgorithm function*

```

async function goalsAlgorithm(userId, goal) {
  try {
    const user = await User.findById(userId); // Fetch the user from the database
    const goalDate = new Date(goal.date);
    const currentDate = new Date();

```

```

//calculates difference in days from now to the end date of goal
const diffTime = Math.abs(goalDate - currentDate);
const diffDays = Math.ceil(diffTime / (1000 * 60 * 60 * 24));
//gets amount to gain and spend because of subs and salaries during this
let subExpense = 0;
for (let sub of user.subs) {
  switch (sub.howoften) {
    case 'daily':
      subExpense += sub.amount * diffDays;
      break;
    case 'weekly':
      subExpense += sub.amount * Math.floor(diffDays / 7);
      break;
    case 'monthly':
      subExpense += sub.amount * Math.floor(diffDays / 30);
      break;
  }
}
let salaryIncome = 0;
for (let salary of user.salaries) {
  switch (salary.howoften) {
    case 'daily':
      salaryIncome += salary.amount * diffDays;
      break;
    case 'weekly':
      salaryIncome += salary.amount * Math.floor(diffDays / 7);
      break;
    case 'monthly':
      salaryIncome += salary.amount * Math.floor(diffDays / 30);
      break;
  }
}
//netincome during the period
const netIncome = salaryIncome - subExpense;
var calcValue = netIncome - goal.amount;
calcValue /= diffDays;
//calcValue is amount to gain every day to meet the goal

goal.calcValue = parseInt(calcValue);
goal.updated = true;

```

```

user.goals.forEach(g => {
  if (g.id === goal.id) {
    g.calcValue = goal.calcValue; //save the amount to the goal obj
    g.updated = true;
  }
});

user.markModified('goals');
await user.save(); //save changes

} catch (err) {
  console.log(err.message);
}
}

```

## Other way to add money to the goal

```

<form action="/income" method="post">
  <label for="amount">Amount</label>
  <input type="number" step="0.01" name="amount" required>
  <label for="date">Date</label>
  <input type="date" name="date" value="<%= new Date().toISOString().split('T')[0] %>" required>
  <label for="category">Category</label>
  <input type="text" name="category" required>
  <label for="description">Description (optional)</label>
  <input type="text" name="description" >
  <!-- this is the incomeaddition form and if there is a goal new parameter to fill appears-->
  <% if (user.goals.length > 0){ %>
    <label for="forgoal">Percentage of income to goal (optional)</label>
    <input type="number" min="1" max="100" name="percentage"/>
  <% } %>
  <!-- this parameter is percentage from income to add to the goal-->
  <button class="btn" type="submit">Add</button>
</form>

```

## Income page

Here I will describe process of adding money to goal using income page.

```
module.exports.income_post = ('/income', async (req, res) => {
  const { date, amount, description, category, percentage } = req.body;
  const newIncome = { date, amount, description, category, percentage};
  try {
    const id = getUserId(req);
    id.then(async (id) => {
      await addIncome(id, newIncome);
      res.status(200).json({ message: 'Income added' });
    });
  } catch (err) {
    console.log(err);
  }
});

AddIncome function
async function addIncome(userId, income) {
  try {
    var newIncome = income;
    const inAmount = income.amount;
    var update;
    const user = await User.findById(userId);
    if (income.percentage == 0) { //if percentage is 0 then add income as it is
      update = {
        budget: (user.budget + parseFloat(income.amount)).toFixed(2),
        $push: {
          incomes: income, // Push the income object to the incomes array
        },
      };
    } else {
      const percentage = income.percentage / 100; //percentage calculation
      const newAmount = parseFloat(income.amount - (income.amount * percentage)).toFixed(2); // this is added to
      budget
      newIncome.amount = newAmount; // Use toFixed() to format the number with 2 decimal places
      const forGoal = parseFloat(inAmount * percentage).toFixed(2); //this is added to goal
      user.goals[0].savedAmount = (user.goals[0].savedAmount || 0) + parseFloat(forGoal); //adding money to the
      goal
      update = {
```

```

    budget: (user.budget + parseFloat(newIncome.amount)).toFixed(2),
    $push: {
      incomes: newIncome, // Push the modified income object to the incomes array
    },
  };
  user.markModified('goals');
}
await user.updateOne(update);
await user.save(); //saving changes
} catch (err) {
  console.log(err.message);
}
}

```

## Overview page sorting

### Overview get

```

module.exports.overview_get = async (req, res) => {
  const userId = await getUserId(req);
  const user = await User.findById(userId);
  const expenses = user.expenses;
  const incomes = user.incomes;
  res.locals.expenses = await mergeSort(expenses);
  res.locals.incomes = await mergeSort(incomes);
  res.render('overview');
}

```

#### *MergeSort, Merge functions*

```

function mergeSort(arr) {
  if (arr.length <= 1) {
    return arr;
  } // If the array has 1 or fewer elements, return the array as it is already sorted

  const mid = Math.floor(arr.length / 2);

```



```

// Calculate the middle index of the array
const left = arr.slice(0, mid);
// Split the array into two halves, left and right
const right = arr.slice(mid);
// Split the array into two halves, left and right
return merge(mergeSort(left), mergeSort(right));
// Recursively call mergeSort on the left and right halves, then merge the sorted halves
}

function merge(left, right) {
  let result = [];
  // Create an empty array to store the merged result
  let i = 0;
  let j = 0;
  // Initialize two pointers, i and j, for the left and right arrays respectively
  while (i < left.length && j < right.length) {
    // Compare the elements at the current positions of the left and right arrays
    if (left[i].date < right[j].date) {
      result.push(left[i]);
      // If the element in the left array is smaller, push it to the result array
      i++;
      // Move the pointer of the left array to the next element
    } else {
      result.push(right[j]);
      // If the element in the right array is smaller, push it to the result array
      j++;
      // Move the pointer of the right array to the next element
    }
  }
  return result.concat(left.slice(i)).concat(right.slice(j));
  // Concatenate the remaining elements of the left and right arrays to the result array
}

```

# Testing

This part will be in the following format:

1. I describe thing I test from the technical solution.
2. I provide you with the name of the video file in **testing** folder.

## Security testing

I will test sign up page and login page. I conduct a test of error messages when incorrect email format or too short password is entered on signup page. As well as that, I will check if new user is created in my database and that the jwt cookie is created. Then I will log out and try to log in. I will test error messages for log in page as well.

Testing of sign-up page, error messages, cookie creation and that user was added to database are in the following file:

File name: auth\_1\_testing

Testing of log out, log in, error messages and sign-up attempt for the same email are in the following file:

File name: auth\_2\_testing

## **Incomes and expenses testing**

I will check if incomes/expenses are added to the arrays in the user object via MongoDB website, will ensure that they are displayed where they should be and check if budget amount changes. Also, I will test that salaries and subs are updated as intended with changing date and new transactions added.

Expense page testing:

File name: expenses\_subs\_testing

Income page testing:

File name: income\_salary\_testing

## **Goal page testing**

I will test that for goal to be created at least 1 sub and 1 salary must be added onto account, that calculate function works as intended, show/hide button and addition to goal.

Goal testing:

File name: goal\_testing

## **Overview page testing**

Overview page is hard to test with insufficient number of incomes and expenses as mergeSort implemented in it is useless when there are few transactions made.

Overview testing:

File name: overview\_testing

# **Evaluation**

## Did I meet the objectives?

### Objectives:

1. A security system that allows for easy user authentication through password protection will be employed. Additionally, session cookies can be encrypted using hashing algorithms as well, thereby ensuring better protection from unauthorized access or any other security breaches while browsing the web.
2. Users should be able to update their data without experiencing any difficulties whatsoever. Thus, keeping all their income and expenses adding easy, quick, and simple becomes essential.
3. The site must be developed such that it manages subscriptions including periodic payments; hence there must be some features on it that remind users about their bills before the day of payment arrives especially for those who cannot remember anything at all about this.
4. Peer-oriented financial g will find a place on this app to guide my course mates by setting your own targets. You can then follow your progress using an interactive chart animation.
5. We will generate personalized reports, that offer a clear picture of financial health to the individuals who wanted deeper insight into their finances, through a system of reporting and analytics.
6. In response to the wish for simplicity that was from all users, an interface that is very user-friendly remains at the top of my agenda.

### Comments:

1. Security system created is robust and fits the objective. Each user is provided with ability to create his own account, passwords are hashed when stored in database providing security of data in case of data leak. Session cookies are also implemented, and work as intended. This objective is 100% met.
2. Income and expense page provides efficient way of adding new transactions onto account. The way of adding new data is simple and straightforward. The objective is completely met.
3. Subscriptions and salaries are implemented on the same income and expenses pages and provide user with immediate way of updating their usual transactions. Monitoring of subscriptions and salaries to be updated on time is also there, so user is provided with full set of tools for tracking their financial state. The objective is met.
4. Users can set personal financial targets and track them via interactive charts/bars. This functionality provides awareness and motivational effect among peers, allowing them to see their progress in real-time. Implementation of the goal tool perfectly aligns with the objective set.
5. Each user has his own personalized page solely for the overview of the financial life of the user. This page is very important and implemented to be as clear as possible for the potential clients. The objective is met.
6. The user interface of the website prioritizes simplicity. The design is intuitive, making interaction as straight forward as possible. By focusing on user-friendliness this objective was perfectly met

## Feedback from users

1. Performance and simplicity feedback:  
“The website loads quickly and is always responsive, also the interface is nice and easy to use.”  
“I liked the way the pages look, very user-friendly interface.”
2. Functionality feedback:  
“The website is very good for tracking incomes and expenses, but the overview page and analytics is not what I expected but still represents overall data efficiently.”  
“Subscription and salary addition is the functionality I wanted specifically, and I was completely satisfied by the way you implemented them, they are very simple and quick in use.”  
“Overall, the website met all my needs but overview page can be expanded into more details”

## Evaluation of feedback

By the user feedback can be seen that overall, the website met all users' needs. But there are several complaints about overview page which point at the small number of details. I agree with this and I think that more work can be done on the analytics part of the website.