Full-Stack Practical: Expense Tracker

Please direct any questions to Feynman (@feynman on Slack).

The goal of this practical is to give us a feel of how you work as well as provide an opportunity for you to demonstrate the core competencies we expect from a full-stack software engineer at Gigster.

We will give you 48 hours for the practical, but please do not feel obligated to spend more than 6 hours. We do not expect perfection, and *the optional requirements truly are optional*.

Description

> *As a consumer, in order to better understand my spending patterns I want to input my expenses in a tracking system which can generate reports.*

Using whatever programming languages and frameworks you are most effective in, build a web application which allows users to keep track of how they spend their money. The deliverable is a full-stack web application, including a back-end server as well as a front-end client.

Requirements

The deliverable should satisfy the following functional requirements:

1. Multiple user accounts should be supported
2. There should be two types of users: **regular users** and **admins**
3. A **regular user**:
    a. Can log in and log out
    b. Can generate **reports** of their spending over time (described in more detail in a later requirement)
    c. Can create, read, update, and delete (CRUD) **expenses** they own
    d. Can *not* CRUD expenses they do not own
4. An **admin**:
    a. Should also satisfy requirements 3(a) through 3(c)
    b. Can read *all* the saved expenses, including those which they do not own
    c. Can *not* create, update, or delete expenses they do not own
5. An **expense**:
    a. Is owned by exactly one **user**
    b. Contains at least the following fields:
        i. Datetime, the date and time the transaction was made
        ii. Amount, the amount of money (in USD, precision of 0.01) associated with the expense
        iii. Description, a string describing the details of th transaction
    c. Can only be created by a logged in user
    d. Can only be read by either the user who owns it or an **admin**
    e. Can only be updated and deleted by the user who owns it
6. A **report**:
    a. Shows the total amount spent per week by the logged in user
    b. Can be filtered to only show expenses occuring within a user-provided datetime range
    c. Can only be generated by users who are logged in
    d. Should not contain expenses not owned by the user (even if the user is an admin)
7. The back-end should provide an interface which is agnostic to any particular front-end client implementation
8. The front-end should be a single-page application which does not need to refresh the browser after initial load
9. Documentation and tests should be provided
10. Software engineering best practices should be followed

Optional

Please don't feel obligated to do any of these unless you have time left over.

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

Evaluation Criteria

We will be evaluating how well your deliverable meets the stated requirements. In addition, the following will also be taken into consideration:

- Code quality
  - Maintainable:
    - Code is easy to understand, free of bugs/typos/unused or commented blocks, and re-used (DRY, don't repeat yourself) where appropriate
    - Uses pre-existing libraries/frameworks where appropriate
    - Tests provide high test coverage and document intended behavior
      - Unit tests should utilize mocks/stubs/fixtures to isolate dependencies (e.g. databases, back-end when testing client code)
      - Integration tests should utilize a headless browser testing framework (e.g. Casper, Selenium) to simulate a real user interacting with the application
    - Documentation is provided for non-trivial functions and classes
  - Modular: functionality is organized into modules which are decoupled and individually testable in isolation
  - Tested: both unit and integrations should be provided
  - Security best practices: sensitive info (e.g. passwords) stored appropriately, authorization on protected resources
- Working style:
  - The candidate is open about their strengths/weaknesses and is not afraid of asking for help
  - Questions or problems blockers encountered while working are quickly escalated to the interviewers
  - Git history is clean (descriptive commit names, semantically related change sets) and easy to follow