Calculator RFC

1. Introduction:

The calculator is a basic tool that is used for performing mathematical operations such as addition, subtraction, multiplication, and division. The purpose of this RFC is to propose the development of a calculator application that is accessible through a web browser.

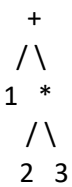
2. Requirements:

a. Screen: to display input and result of calculation
b. Buttons:
    i. Number pad with digits 0-9 and decimal point
    ii. Binary operators: addition, subtraction, multiplication, division, exponential
    iii. Unary operators: square root, percentage, (unary minus sign)
    iv. Memory functions: M+, M-, MR, MC
    v. History: display previous calculations
c. Navigation bar display authentication status and options to sign up/sign in
d. Logic:
    i. Should support all valid calculations with the given buttons.
    ii. Can be used with or without authentication.
    iii. Follow order of operations (PEDMAS)

3. Technical architecture:

To illustrate proficiency in Python and TypeScript, the frontend is written with TypeScript using React and MaterialUI components. The backend is a simple Flask Python server that handles basic authentication REST requests with a simple

a. User authentication: React frontend passes login/signup information to Python backend for persistent storage and abstracting authentication logic
b. Expression parser:
    We need to parse an expression “1 + 2 * 3” into a structure that is traversable while keeping the order of operation intact. There are quite a number of different ways to do this, but the easiest is to build and Abstract Syntax Tree, which is just a flavor of binary trees that can be recurse on.



    It’s quite simple to evaluate the tree once we have it, the less intuitive logic is to parse a string into such a tree. We need to define a parsing grammar.
    Intuitively, it would be something like

```
EXP -> EXP + EXP | EXP - EXP | EXP * EXP | EXP / EXP | - EXP | (EXP) | number
```

The problem with this is that it doesn't preserve the correct ordering (we'd like '*' to have higher priority than '+'). Also, it doesn't know when to terminate, so we are stuck in a recursive loop. We can fix this by introducing different types of EXP and tweak our grammar.

```
EXP    -> TERM EXP1
EXP1   -> + TERM EXP1 |
          - TERM EXP1 |
          epsilon
TERM   -> FACTOR TERM1
TERM1  -> * FACTOR TERM1 |
          / FACTOR TERM1 |
          epsilon
FACTOR -> ( EXP ) | - EXP | number
```

This is much better, everything terminates and we have our preferred order of execution. Note that epsilon is a terminal expression, or EOF. Extra operators are based on this logic.

c. History and memory functions are implemented as states in React. The logic is too simple to pass to backend for processing.

4. Drawbacks / Further work

a. Password is passed unencrypted through network and stored unencrypted in database

It's known that this is a huge security risk, especially if HTTP requests are intercepted or data tables are compromised. (I don't have time to implement session token-based authentication)

b. Expression parser

Without next token lookup, we can't support implicit multiplication like (2)(2) or 2sqrt(2). We can rewrite with a smarter parser that can look forward a few tokens. There is also edge cases that remains to be unhandled like division by zero.

c. Saving and restoring user session:

It should be very easy to create new data table in Flask backend to store previous calculations and restore them upon login. We already have a "History" interface.