

# Design Details (by Mridul Jain)

## *Aim*

To make an intelligent this-or-that chart webapp for investment strategies.

## *Description*

Given a dataset consisting of performance of 'n' strategies, we want to show the strategies two at a time to the user. The performance set consists of the profit and loss series of the strategies for the last 't' years. The user makes a call whether he likes the first strategy or the other.

So lets say given the data for  $n=100$  strategies for the past  $t=20$  years, we want to build an application which can display interactive charts for each of the strategy and the user can select either of them to see more details about them, like the sharpe ratio etc., described in Appendix attached.

## *The Collaborative Mixin*

Since we are already getting some pretty useful information from the user, we would like the app to be a little intelligent. Based on some choices of the user, we would like to show strategies to the user where we are more sure of the user's choice and make recommendations to new users based on choices of previous users. You can ask the user for some information as well (maybe as small popups/tooltips).

## *The ToDo List*

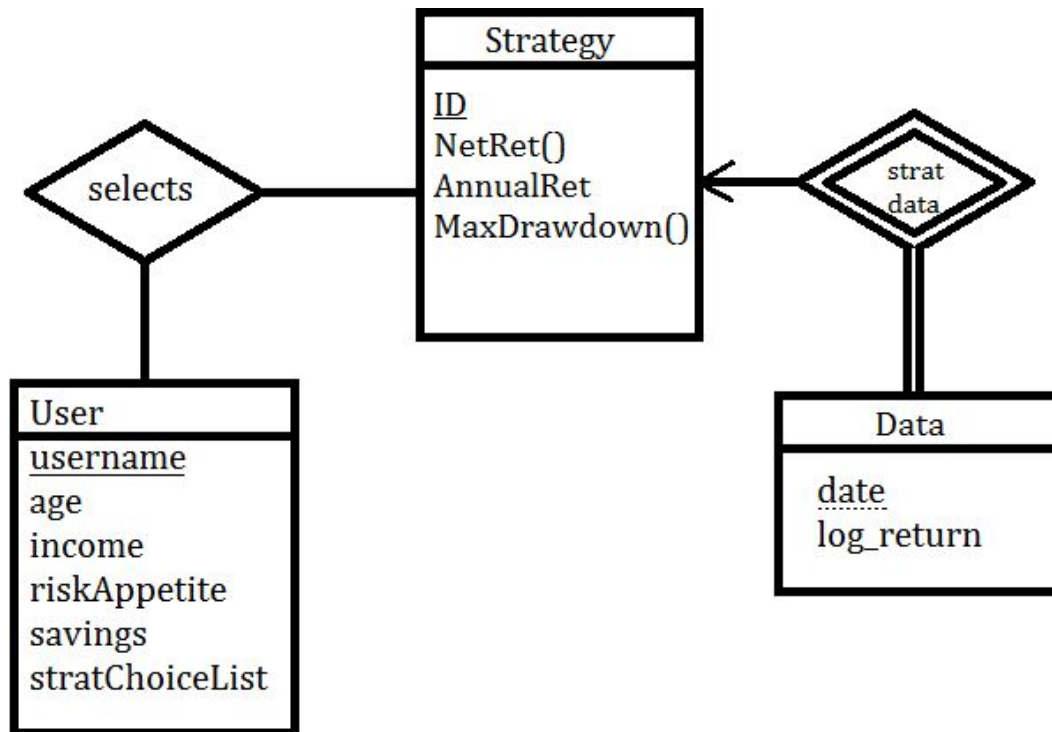
- Setup a backend server to record user responses, serve strategy data.
- Develop a beautiful, responsive and high-performance UI to showcase strategies and gather user choices.
- Record user choices, display the next pair based on user's previous choices.
- For a new user, gather information about him and then use that information to make suggestions.

### Backend design:

For this application I intend to set-up a backend server to record user responses and serve strategy data. This is basically a database application and thus I need :

1. To design the database schema
2. To write functions that access and update this data (since in the application we are taking user input as the only input data so we need to write function that updates the user data; strategy data is only accessed but not changed/updated at any point of time)

Based on the application requirements I identified the following entities and relationships between them.



The above E-R diagram can be converted into the following relational database schema with 4 relations in our database (with comma separated attributes in each relation, primary keys underlined):

1. strategy (ID, NetRet(), AnnualRet(), MaxDrawdown() )
  - a. ID : unique name given to the strategy
  - b. NetRet: a derived attribute showing net returns of this strategy for the simulation period
  - c. AnnualRet: a derived attribute showing Annualized returns for this strategy for the simulation period

- d. MaxDrawdown: a derived attribute showing the largest peak-to-trough decline in the value of a portfolio for this strategy
2. data (strategyID, date, log\_return): the daily log return time series data of 149 strats. This is self-explanatory. strategyID is a Foreign Key that references the strategy table's Primary Key. The strategyID and date together form the primary key for this relation.
3. selects(strategyID, username): an entry (sID, uID) is created in this table when a user (with username uID) selects a strategy(with ID sID)
4. user (username, age, income, riskAppetite, savings, stratChoiceList):
  - a. username: unique username created for each user when he sign-ups for using the app
  - b. age: user's exact age (integer).
  - c. income: annual income range(in LPA) of the user (single precision)
  - d. riskAppetite: in terms of percentage of the money invested (integer)
  - e. savings: in terms of percentage of the money saved (integer)

While every other attribute here is self-explanatory, stratChoiceList needs some explanation. stratChoiceList is a NOT simply the list of user choices so far. It is actually a list of strategies which shows the order in which strategies would be shown to the user. We can think of this like a queue.

The most important part of our application is a function which takes as input a single strategy selected by the user and returns a pair of strategies which should be shown next to the user. We can think of it like a knockout tournament to find a winner. This can be modeled as the level order traversal of a binary tree from bottom-to-top. A level-order traversal can be implemented by a queue data structure as follows:

```

create an empty queue Q
push all the strategies in Q (the order in which strategies are
pushed is determined based on other user parameters explained
below)
while(Q.size() > 1)
{
    s1 = Q.pop()
    s2 = Q.pop()
    Display(s1,s2) to the user
    Let user select a strategy(say s) from among s1 and s2
    Q.push(s)
}

```

When queue size becomes 1, we have only a single element inside the queue and that is the winner strategy. stratChoiceList is the Q as in the above pseudo code.

**The order in which strategies are pushed in the queue initially** is determined based on other user parameters: age, income, riskAppetite and savings. Based on these user preferences we want to determine an order among the strategies from the most favorable strategy to the least favorable one (I will explain how I determine this order in the next paragraph). Let's say we have 8 strategies and we have determined an order: 1,2,3,4,5,6,7,8 with strategy 1 being the most favorable and strategy 8 being the least favorable. Then we push them in this order in the queue: 1,8,2,7,3,6,4,5 i.e. most favorable, least favorable, next most favorable, next least favorable and so on.

This is because we want the most favorable strategies to make it to the next round (this is most favorable based on user preferences and thus we want this to win the tournament whereas if we compare two very strong strategies right in the beginning then a strong strategy is bound to be eliminated earlier) and this can be done only if we compare it to the least favorable strategy. Once we compare the most favorable and least favorable, we apply the same principle to remaining strategies and get the required order.

**To determine the order of strategies from most favorable to least favorable:** Based on the entries in *selects* table we can identify the users that have chosen a particular strategy previously. With every user we already have stored attributes: age, income, riskAppetite, and savings. Using this information, I will follow the following procedure for obtaining the overall order of preference:

1. For every strategy
    - a. Get the list of users that have chosen this strategy previously (call it prevUser)
    - b.  $(\sum \text{abs}(\text{thisUser.age} - \text{prevUser}[i].\text{age}))/N$  where  $N = \text{size}(\text{prevUser})$
    - c.  $(\sum \text{abs}(\text{thisUser.income} - \text{prevUser}[i].\text{income}))/N$
    - d.  $(\sum \text{abs}(\text{thisUser.riskAppetite} - \text{prevUser}[i].\text{riskAppetite}))/N$
    - e.  $(\sum \text{abs}(\text{thisUser.savings} - \text{prevUser}[i].\text{savings}))/N$
  2. Sort the strategies in order parameter b above. This will assign a rank to each strategy
- Similarly,
3. sort the strategies in order of parameter c above: get a (possibly) different ranklist
  4. sort the strategies in order of parameter d above: get a (possibly) different ranklist
  5. sort the strategies in order of parameter e above: get a (possibly) different ranklist

Finally we have 4 ranks for every strategy. We take an average of all the 4 ranks and get a final rank(possibly non-integer) for each strategy. We then sort all the strategies in the order of these ranks and get the overall order of preference from most favorable to least favorable.

All of the above procedure can be summarised into following functions:

1. **userInput:** A function that takes as parameter a username and strategy ID, returns with 2 strategy IDs (explained above)
2. **login:** function that takes as parameter a username which checks if user is registered by querying the **user** table. If user is registered it returns with a 2 strategy IDs

3. **signUp**: function that takes as parameter a username and other parameters and creates an entry in the **user** table (also populates the stratChoiceList attribute based on user parameters)
4. **fetchData**: A function that takes as parameter a strategy ID and returns a list of (date,daily\_log\_returns), NetRet, AnnualRet, MaxDrawdown. This will be used by the UI.

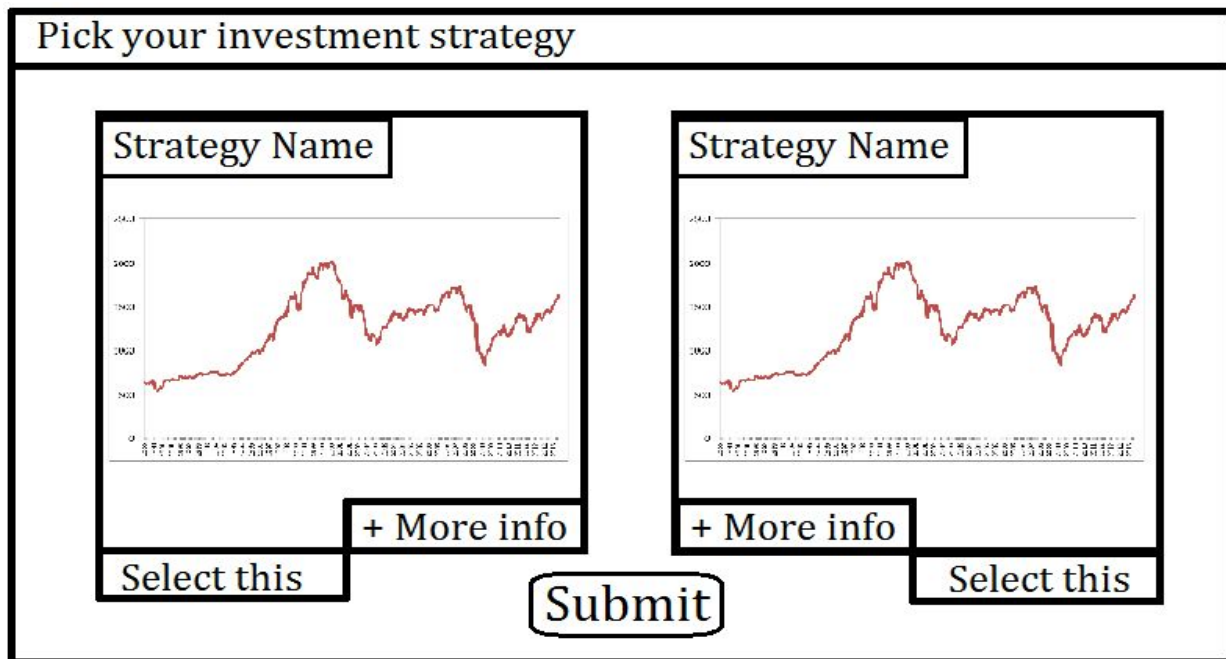
### *Frontend design:*

I intend to build an intuitive and high-performance UI to showcase strategies and gather user choices using React JavaScript library (<https://facebook.github.io/react/index.html>). React is all about modular, composable components. In this we simply express how our app should look at any given point in time, and React will automatically manage all UI updates when our underlying data changes.

We proceed to design the UI in a step-wise manner

(<https://facebook.github.io/react/docs/thinking-in-react.html>) :

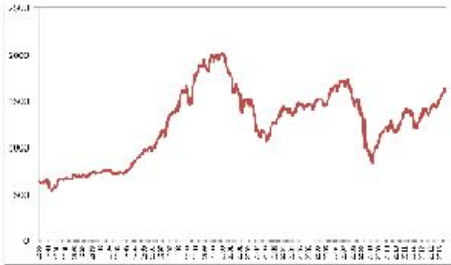
1. Start with a mock design



2. Break the UI into a component hierarchy

### Pick your investment strategy

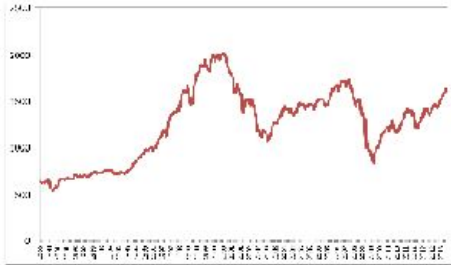
Strategy Name



+ More info

Select this

Strategy Name



+ More info

Select this

Submit

- AppBody (Yellow)
    - Option (Red)
    - Option (Red)
    - Submit (Green)
3. Build a static version in React
- AppBody (props: username)
    - Option(props: ID1)
    - Option(props: ID2)
    - Submit(props: callback function onSubmit(ID))
4. Identify the minimal (but complete) representation of UI state and where the state should live
- AppBody (state: ID1, ID2)
    - Option(state: list of (date,daily\_log\_returns), NetRet, AnnualRet, MaxDrawdown)
    - Option(state: list of (date,daily\_log\_returns), NetRet, AnnualRet, MaxDrawdown)
5. Add inverse data flow
- We need to pass data from the child component 'Submit' back up to its parent 'AppBody'. We do this in our parent's render method by passing a new callback (handleSubmit) into the child, binding it to the child's onSubmit event. Whenever the event is triggered, the callback will be invoked with the ID of selected strategy as the

parameter. This callback function will in turn call `fetchData` function on the server side by passing the username and ID of selected strategy as the parameters. Once `fetchData` returns a pair of strategies(`ID1'`, `ID2'`) we will modify the `AppBody`'s state as `this.setState(ID1', ID2' )`.

This broadly sums up the frontend design of the app. The above aspects of UI design may seem a little vague at the moment. The things will be more clear once I start the implementation.