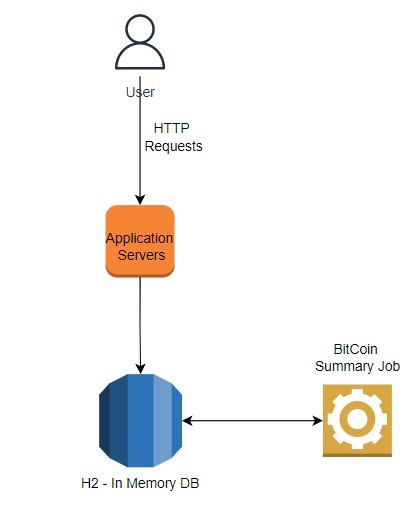
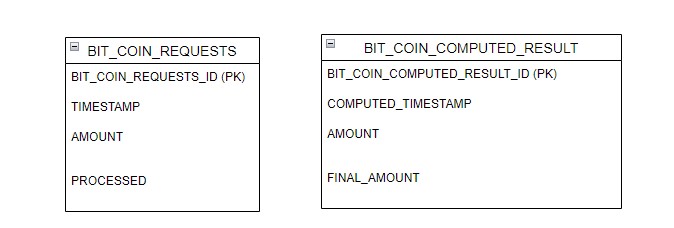
# Design Document

The design is simple. A rest API is exposed that helps to handle the requests from the user A in-memory database H2 is used to store the data received from the users. A separate bit coin summary job is run every minute (configurable cron schedule) and it retrieves the bit coin requests record from the database computes the amount received per hour and saves them in another table. The cron schedule will be responsible for the computation of the final amount and the accuracy of the result will depend on how frequent the summary job is run. Here I believe, the final amount compute need not be accurate and an eventual consistency is enough.



The table structure are as follows



BIT\_COIN\_REQUESTS stores the requests as they reach the server. BIT\_COIN\_COMPUTED\_RESULT stores the computed result. There are two columns for storing the amount. AMOUNT and FINAL\_AMOUT. AMOUNT stores the amount for that particular hour, FINAL\_AMOUNT stores the amount received util that hour.

### Design Considerations & Improvements

* The database used is in-memory and the data is not persisted after application reboot. To solve this, we can use a database like MySQL or any relational database
* Only one application server is present. In order to scale we need to setup multiple application servers behind a load balancer.
* There is no authentication supported. In order to support authentication, we can integrate with a separate authentication service.
* The application will not scale if there are many simultaneous requests received. In order to address them, we can use distributed databases to store our requests or alternatively send the requests received to a Kafka like messaging queue and another distributed processor computes the results and stores it in a Redis or mem-cache as well as in a persistent database.
* What happens to the request that are received in out of order? Again, it depends on our compute logic. Two approaches
  + Discard the request
  + Process the out of order messages by a separate processor and update the historical record in a passive way.

Considering all the factors the final high level design diagram is given below

* MySQL is used for persistence (Other distributed databases can be used as well).
* The database is replicated prevent single point of failure.
* In order to scale to multiple servers, the database can be sharded based on generated key.
* We can use twitter-snowflake to generate an id (PK) that is unique across all the application servers.
* Once the request is received, the entry is saved in database as well as sent to a Kafka queue
* A consumer consumes from the Kafka and computes the result and stores in cache for easy retrieval and access.
* Final computed result is stored in database for persistence
* The application servers will retrieve the compute results from Redis and serves them to the user.
* SSL termination is done at the load balancer
* Each request is authenticated through the authentication request. The flow can be, the load balancer forwards the requests to the authentication service, only requests that are successfully authenticated will be sent to the application servers.

