# Fun With Banking

The following project provides an opportunity to demonstrate some of the database, infrastructure and system skills required to be successful at Xcalar. This is an “open-book” project and you are encouraged to use online resources as necessary, however we do ask that this be your own work. Given familiarity with all the technologies involved this should take about three hours to complete. Feel free to spend more time if some of the underlying technologies (eg docker or faker) are unfamiliar.

In this project you will be providing a simple transactional banking system built around **Postgres**. Postgres will be run in a docker container and will expose a server on a port mapped into the host system. You will write a python script which will run **outside** the docker container and connect to the Postgres server using **ODBC** via the exposed port. Your python script will populate the database with realistic mock data. From your python script you will run several queries against these data.

There are two main entities in this database: **customers** and **accounts**. A **customer** has the following attributes: unique customer ID, first name, last name, street address, city, state, zip and creation timestamp. **Accounts** have only three attributes: unique account ID, balance and creation timestamp. The relationship between customers and accounts is many-to-many. Customers can have multiple accounts, and accounts can be shared across multiple customers. You might need to introduce a third table to capture this relationship. Please use no more than three tables. You should assume there will be concurrent users of this database. Since this is just a demo, don’t worry about security; passwordless ODBC is fine.

**If you get stuck please simplify the problem to make progress**, even if that deviates from the requirements. **Look at the objectives on the next page** and try to fit any deviations into these objectives. If you’re unable to complete all parts of the project please still submit what you have.

Please see the bulleted lists on the next page for further details.

**Objectives, demonstrate:**

- Interpreting and meeting requirements independently
- Delivering a working end-to-end system
- Working with several virtualized environments (docker, virtualenv)
- Generating realistic test data
- Basic python coding ability
- Understanding of non-trivial SQL
- Basic understanding of database transactions

**Questions (please answer in README):**

- How much time did you spend on this?
- Did you receive outside help (eg a friend, colleague, forum posting, etc)?
- What were the most useful web pages (URLs) in completing this project?
- What new technologies did you have to learn to complete this?
- Do you believe your results are correct? Why?
- What are the biggest gaps between what you delivered and the requirements?

**Deliverables (six files in tar archive named <firstname>_<lastname>.tgz; less than 100KB):**

- **README:** Things we should know about your project (**your full name**, how to run it, notes, etc)
- **Dockerfile**: Around 5 lines
  - Defines docker image with Postgres database
  - Automatically load schema on container startup
- **init.sql**: Sets up the initial Postgres schema in docker
- **requirements.txt**: Output of "`pip freeze`" in your python virtualenv
- **setup.sh**: A few lines to build and run the docker image
- **test.py**: ~300 lines; main test script
  - Runs in **virtualenv outside the docker** on the host
  - Use **`pyodbc`** to connect to Postgres running in the docker
  - Populate DB with mock test data and run demonstration SQL queries
    - Recommend using **`Faker`** and **`random`** to generate test data but feel free to use something else
    - Use a seed to make test data generation deterministic
      - Must be **idempotent**: produce the same output for a given seed each run
    - Randomize number of accounts generated per customer
  - Script arguments (add more if desired)
    - Parameterize number of customers, default to 1,000
    - Parameterize max number accounts generated per customer, default to 10

**SQL Queries (run from test.py over ODBC):**

- Find the states having the 10 highest average customer balances
- Find the 10 customers with the highest total balances
- Find the 10 customers with the lowest total balances
- Operation Robinhood: transactionally transfer 10 percent of each of the top 10 customers largest account to the each of the bottom 10 customers