# CPSC 449 Project 1

In this and subsequent projects we’re going to build services for a web application similar to [reddit](https://www.reddit.com/).

**DUE MARCH 20th**

## Development

For this project you will **build two microservices** for specific functionality of this site and **two automation test suites** for these services.

### Services

#### Posting microservice

Each post should have a title, text, a community ([subreddit](https://www.dictionary.com/e/slang/subreddit/)), an optional URL linking to a resource (e.g. a news article or picture), a username, and a date the post was made.

The following operations should be exposed:

* Create a new post
* Delete an existing post
* Retrieve an existing post
* List the *n* most recent posts to a particular community
* List the *n* most recent posts to any community

When retrieving lists of posts, do not include the text or resource URL for the post.

#### Voting microservice

Each post maintained by the posting microservice can be voted up or down. This service should maintain the number of upvotes and downvotes for each post. A post’s score can be computed by subtracting the number of downvotes from the number of upvotes.

The following operations should be exposed:

* Upvote a post
* Downvote a post
* Report the number of upvotes and downvotes for a post
* List the *n* top-scoring posts to any community
* Given a list of post identifiers, return the list sorted by score.

Each upvote or downvote should include a unique identifier (e.g., a URL or database key) for the post that can be used to match votes with the posts maintained by the posting microservice.

If this service is implemented with a database separate from the posting microservice, it is not responsible for verifying the existence of a post before recording or reporting votes.

#### User Account Microservice

Each user who registers should have the following data associated with them:

* Username
* Email
* Karma

The following operations will be exposed:

* Create user
* Update email
* Increment Karma
* Decrement Karma
* Deactivate account

The data for the user can be in the same database or different database as the other services.

#### User messaging Microservice

Users can send and receive messages to each other. Messages will consist of the following data associated with them:

* Message ID
* User from
* User to
* Message timestamp
* Message contents
* Message flag

The following operations will be exposed:

* Send message
* Delete message
* Favorite message

Messaging data can be in the same database as other services or a separate one.

### API implementation

Implement your APIs in Python 3 using [Flask](https://flask.palletsprojects.com/en/1.1.x/). You are encouraged, but not required, to use [Flask API](https://www.flaskapi.org/) to obtain additional functionality.

All data, including error messages, should be in JSON format with the Content-Type header field set to application/json.

### HTTP status codes

Use appropriate HTTP status codes for each operation, with the following guidelines:

* In general, successful operations other than POST should return HTTP 200 OK.
* A successful POST should return HTTP 201 Created, with the URL of the newly-created object in the Location header field.
* Attempts to retrieve or modify an existing object should return HTTP 404 Not Found if the specified object does not exist (or no longer exists). Note that this does not apply to objects maintained by other services.
* Operations which result in a constraint violation such as attempting to INSERT a duplicate value into a column declared UNIQUE or attempting to INSERT a row with a FOREIGN KEY referencing an item that does not exist in another table should return HTTP 409 Conflict.

### Session state

Requests to each microservice must include all information necessary to complete the request; your APIs must not use the Flask session object to maintain state between requests.

### Database

Use The Python Standard Library’s [sqlite3](https://docs.python.org/3.6/library/sqlite3.html) module as the database for your Flask application. You may use separate databases for each Flask application, or share a database across microservices.

## Testing and Automation

### Basic validation testing

Each microservice should have an accompanying test script to verify that your newly-defined API endpoints work correctly and to populate the microservices with some sample data. Suitable approaches to scripting include:

* A shell script that calls [curl](https://alvinalexander.com/web/using-curl-scripts-to-test-restful-web-services) commands

Note that it is complicated to [determine whether a curl command has succeeded](https://stackoverflow.com/questions/38905489/how-to-check-if-curl-was-successful-and-print-a-message) programmatically, so you will probably need to read the output carefully.

* A Python script using the [Requests](https://realpython.com/api-integration-in-python/) library

Use the following command to install Requests on Tuffix:

$ **pip3 install --user requests**

* A YAML script using the [Tavern](https://taverntesting.github.io) plugin for [pytest](https://pytest.org/)

Use the following command to install Tavern on Tuffix:

$ **pip3 install --user tavern**

### System Testing

In addition to basic testing, each group should come up with a framework to test it’s services with load, and in simulated user-scenarios. Multiple users should be able to be simulated concurrently. Each group may pick whatever frameworks and automation tools will be needed to test and simulate this.

Your system test suite must:

* Test all services
* Perform a load test simulating 100 users
* Simulate a real user scenario using the two services
* Stress the service with: excessive load, bogus data, negative tests

## Operations

You may use any platform to develop services and tests, but the test environment for projects in this course is a [Tuffix VM](http://csufcs.com/tuffixinstall) with [Python 3.6.7](https://www.python.org/downloads/release/python-367/). It is the responsibility of the Operations role to ensure that your code runs on this platform, scales, and can be deployed for production in this environment . As part of the deliverable the operations role will deliver **runbooks to deploy, setup, run, and scale your services, and run your tests.**

### Runbooks

Runbooks are the used by your operations team to setup, maintain, and run your applications. They should include steps to provision and setup your application and required components. They should include the steps to maintain and upgrade components, scale, restart, any tasks that are needed for the application.

See samples and references:

<https://medium.com/@shawnstafford/ops-runbook-16017fa78733>

<https://www.ibm.com/garage/method/practices/manage/operationalize-app-readiness/runbooks-to-automate-operations>

<https://wa.aws.amazon.com/wat.concept.runbook.en.html>

### WSGI Server

While the developers will use the [Flask Development Server](https://flask.palletsprojects.com/en/1.1.x/server/) to run and test their code, in production the code will need to be [deployed to a WSGI server](https://flask.palletsprojects.com/en/1.1.x/deploying/wsgi-standalone/#gunicorn) such as [Gunicorn](https://gunicorn.org/). To install Gunicorn for Python 3 on Tuffix, use the following command:

$ **sudo apt install --yes gunicorn3**

Be sure to run Gunicorn with the gunicorn3 command rather than gunicorn as shown in the documentation.

### Managing processes

Write a Procfile and use [foreman](https://ddollar.github.io/foreman/) to start and manage the Gunicorn and Caddy processes for each microservice. Configure Gunicorn to listen on the port specified by the $PORT environment variable so that Foreman can manage the assigned ports.

### Load balancing

Use foreman start -c to start 3 instances of each microservice. (See *Advanced Options* in [Introducing Foreman](http://blog.daviddollar.org/2011/05/06/introducing-foreman.html) for details, but note that the option has changed to [-m or --formation](https://ddollar.github.io/foreman/#RUNNING) if you are running a more recent version.)

Install version 1 of the Caddy web server using the following command:

$ **curl https://getcaddy.com | bash -s personal**

Create a [Caddyfile](https://caddyserver.com/v1/docs/http-caddyfile) and use the [proxy](https://caddyserver.com/v1/docs/proxy) directive to do following:

Direct requests for http://localhost:2015/posts to the Posting microservice and requests for http://localhost:2015/votes to the Voting microservice.

Load balance requests for each microservice between the instances of each microservice.

If you are in the same directory as the Caddyfile, start Caddy with the following command:

$ **ulimit -n 8192 && caddy**

Check that when you make repeated requests to the same URL they are directed to different upstream Gunicorn instances.

## Tips

If you are not using PugSQL, read [Using SQLite 3 with Flask](http://flask.pocoo.org/docs/1.0/patterns/sqlite3/) carefully.

Use the Flask [development environment](http://flask.pocoo.org/docs/1.0/server/).

Enable [foreign key support](https://www.sqlite.org/foreignkeys.html#fk_enable) in SQLite

For simplicity, avoid Flask Blueprints and Python packages; implement each microservice in a single .py file.

Get comfortable with the curl and sqlite3 command-line interfaces.

## Teams

This project is written for teams of 3 or 4, randomly selected by titanium. This project may not be submitted individually.

### Roles

Teams must agree on a role for each member, and each project will specify a set of responsibilities for each role. There are two *Development* roles, *SDET*, and one *Operations* role for each project.

### Responsibilities

**Dev 1:** owns development and testing of two of the four microservices.

**SDET**: owns the testing and automation,

**Ops:** owns the runbooks, automation and technology stack for production (choose the packages, database, WSGI server, load balancer) on Tuffix deployment.

Each team member is responsible for documenting their work and assisting other team members with integrating their work together.

## Submission

Submit your Python code, SQL schema, test scripts, Procfile, Caddyfile, documentation, runbooks, and any other relevant artifacts to the project1 assignment in titanium (group submission, only one is required to submit).

## Presentation/Whiteboarding

Your group will whiteboard and architect out it’s solutions and design choices on the due date of the project. For this you will go through how you approached the solution, what components used, any technical solutions that seem interesting (how DB was used, not used for example). Each person will explain their role and work they did.