Lab 3
CS-554
Designing Technology Stacks

Scenario 1: Logging

This scenario doesn't specify a specific client technology; my logging server will thus be language agnostic, allowing many different applications to log to it. To achieve this, logging entries will be done via a web server, and then stored in a backend database. The webserver will also serve as a front end, allowing users to view and query their log entries.

For the web server portion of the stack, I will use **nginx**. Users will submit log entries to the server via HTTP requests, allowing virtually any application to submit logs to the server.

The main application will use **Node.js**, as it is a capable server side language that excels in asynchronous applications, and has a rich drivers for other technologies. Specifically, node has a driver for postgresql (**node-postgres**), which will be the backend data store for this application.

In the front end, we will be using **express** and **express-session** for serving web pagessession management and **React.js** for the user interactive web pages. We will use **node-postgres** to connect to our database in the back end.

This stack will use **PostgreSQL** for data storage. This application will be best served by a relational database, as users will typically be querying the data by a fixed set of fields. Fields in the relational database will include fields such as "server", "loglevel", and "timestamp". Log entries are stored as varchar, which can also be filtered on. PostgreSQL will also be used as a store for user accounts; it is important that log entries are only accessed by the user that owns the data. If we instead wanted to support more unstructured data, we could instead consider **mongodb**; this, however, is strongly depended on what kind of data we are looking to store.

From a high level, this is a simple yet straightforward stack. Users will first have to create an account on our server. With an account, the user can create API keys for their applications to authenticate with the server.

To log to our server, the client will send an HTTP request to **nginx**. This request will be passed along to the **Node.js** application. Node will validate the request using the API key, validating it against **PostgreSQL** using the **node-postgres** connector. Valid requests will be pushed into the SQL database.

To check log entries, the user will log back into the ngingx webserver. Upon authentication, the user is provided with an interactive web page powered by **React.js**. The user can select which log they would like to view (if they have multiple they are authorized to access). They will be provided with an array of parameters they can query and filter on (for instance, timestamp or log level). In addition, Node.js+React.js will automatically update this data as new log entries are made to the server, allowing the end user to have a live stream of logging events.

All technologies described are *Operating System independent*; **Windows** and **Linux** are both acceptable.

Scenario 2: Expense Reports

For this application, **nginx** will once again be used. This application is expected to be lightweight, so the heavier resource requirements of Apache can be avoided. However, Apache could be sufficient, as well.

For our database, we will once again employ a relational database. The data in the spec is strictly of a fixed, unchanging structure, so the choice is clear. There are a number of FOSS SQL databases that will largely be equivalent of our small application; for this reason, we will once again simply select **PostgreSQL**.

For our website itself, we will use **Python** as our main server code, gluing our front and back end together. Python is selected for this purpose due to its ease of use, ease of extensibility, wide variety of libraries including database connectors and front end http/css libraries.

Brining this all together, our web site will be implemented using Python **Flask**. Flask is a quick, lightweight web framework that will allow us to easily connect to the rest of our tech stack. We will use **flask-login** to manage user authentication. Presumably, user accounts will be stored elsewhere in a company database, that flask with authenticate against; this application will not store user account data.

The front end of this application will be templated using **handlebars** and **bootstrap**, both of which have Python libraries/apis. This will allow us to easily create maintainable, modern looking front end pages/forms for the user to submit their expenses. These tools will then also be used in conjunction with **flask-wkhtmltopdf**, which is the python connector for webkit html to pdf. We can use handlebars and bootstrap to create professional looking html, and wbhtmltopdf to generate our output pdf files.

Finally, the application will send emails using the company smtp server. Presuming this expense reporting is part of a larger company, there will already be email services available. However, we can use our Python environment to connect to and send emails using the existing SMTP server, using Python **smtplib**.

If there is not an existing SMTP server set up, we can set up one simply on any modern **Linux** distribution. A simple SMTP service can be installed on Debian based systems using the aperture package manager.

Scenario 3: Twitter Streaming Safety Service

This service is expected to be a bit heavier weight with more user traffic. This application will utilize **Apache** for is greater performance when scaled, and some of its power module plug-ins. Security is paramount in a safety application, so we will employ the module **mod_evasive**; this is an Apache module designed to protect the application from brute-force and DDoS attacks.

For email, we will use **iRedMail**, a FOSS email client. Not only is this free for us to include in the stack, iRedMail supports SMTP/POP3/IMAP, so we can programmatically send emails easily. iRedMail allows us to create multiple different user accounts as well, allowing us to fine tune the email aspect of this Twitter service, sending different emails to different users depending on configurable settings (keywords, preferences, severity, etc).

For text alerts, we will use **Twilio**. Twilio is a service that allows us to send SMS messages programmatically, and has APIs for many popular languages. Twilio also allows programmatic sending of MMS messages, which could be an extremely valuable if not requisite feature; that is, the service could send a message to specific subsets of officers (ie those on duty), or all officers in the department, or all officers in a specific area, etc.

For our database, **MongoDB** is employed. The primary data of interest in a tweet is the content of the tweet itself, which is by definition unstructured. There will also be fields for the user, the location, and the timestamp of the tweet. In addition, this application must store *all* historical tweets, for future analysis and data mining. By default, MongoDB now comes with the **WiredTiger** storage engine. WiredTiger supports compression for all of our collections, an important feature for what may presumably be a large and ever growing database.

To provide a live stream of tweets/incidents to a user, the application will utilize **WebSockets**. A similar approach would be to use AJAX requests with a short timer, but that would not technically be “streaming”, and would not meet our defined requirements. The webpage and client would have a WebSocket connection, and the webserver will send updates as they are parsed and identified.

Raw file storage will be done using a standard file system on a **RAID** array. RAID provides us with protection against disk failure and file corruption through redundancy. We can also take advantage of striping, to improve performance. Files that are uploaded will be compressed using **ImageMagick**, as files such as images and videos will take up a lot of disk space. Files will be assigned random names for security, and the location/name of the files will be stored in a very lightweight and fast **PostgreSQL** database (because there are fixed fields).

The Twitter API we will use in **Filtered stream**. This API meets all of our requirements, including geolocation rules as well as keyword rules; we can target tweets specifically in our area that hit our keywords, and the API supports Boolean operators to join our rules together.

Finally, we will build a basic **Node.js**+**express**+**express-session**+**ws** web framework to manage our webpage forms, user sessions, and streaming web page. Node will also be dispatch our email and SMS services, and manage IO from our database and file systems.

Once again, this is *Operating System independent*; **Windows** and **Linux** are both acceptable.

Scenario 4: Mildly Interesting Mobile Application

This application does not require any of the heavy weight features of Apache, and once again utilizes **nginx.** Nginx will serve both as our endpoint for users to send HTTP requests to, as well as provide us a location for the administrator dashboard.

Node.js will be used as the web framework, handling the requests from nginx, as well as providing our administration page; for web requests, we will use **express**. To support CRUD, we must also support user accounts. To manage user logins, the stack will user **express-session**. The website will provide all of the CRUD functionality, but this functionality can also be exposed in the API via an API key for each user.

For cheap long term storage, the best approach will utilize a large **RAID** array on a standard **RAID** friendly filesystem (such as **butrfs** or **zfs**). Uploaded files will be pre-processsed using **ImageMagick**. For speed, pictures will not be gzip compressed, but they may be resized or potentially changed to a lossy image type (jpg). The amount of compression can be configured by the end user or controlled by the administrative users. Files will also be assigned random names, for security purposes. The type of RAID employed will also utilize striping, allowing for even faster file reading.

To improve speed, we can utilize a **content delivery network**, which will cache images in their relative geospatial regions. For this application, we will utilize **Cloudinary**, a CDN that specializes in images and video.

File metadata will be stored in a relational database, **PostgreSQL**. This database will store data such as geospatial data (latitude and longitude), timestamps, filenames and paths, and user information.