Scenario 1: Logging

In this scenario, you are tasked with creating a logging server for any number of other arbitrary pieces of technologies.

Your logs should have some common fields, but support any number of customizable fields for an individual log entry. You should be able to effectively query them based on any of these fields.

How would you store your log entries? How would you allow users to submit log entries? How would you allow them to query log entries? How would you allow them to see their log entries? What would be your web server?

We would use **MongoDB, Express and NodeJS** to achieve this task. The logs would not go to console log since that would not achieve the purpose of storing or querying.

The nodeJS application would run on an express server that would expose a POST url: http://servername/addTolog with the following json format:

```
{
   "sender":"sending system",
   "tsp":"date timestamp",
   "file":"source filename",
   "sender_uuid":"thread id/UUID",
   "logging_level":"INFO",
   "message":"This is a sample logging message",
   "optional_fields":[
      {"field_name_0":"field_value_0"},
      {"field_name_1":"field_value_1"}
   ]
}
```

This information would then be stored in the MongoDB (we can use a relational DB with the predefined column names and two tables – one listing the first 4 entries, and the other with a foreign key constraint listing the optional fields). The logging server would create its own UUID before inserting the data into the database.

The querying part would be handled by various GET requests:

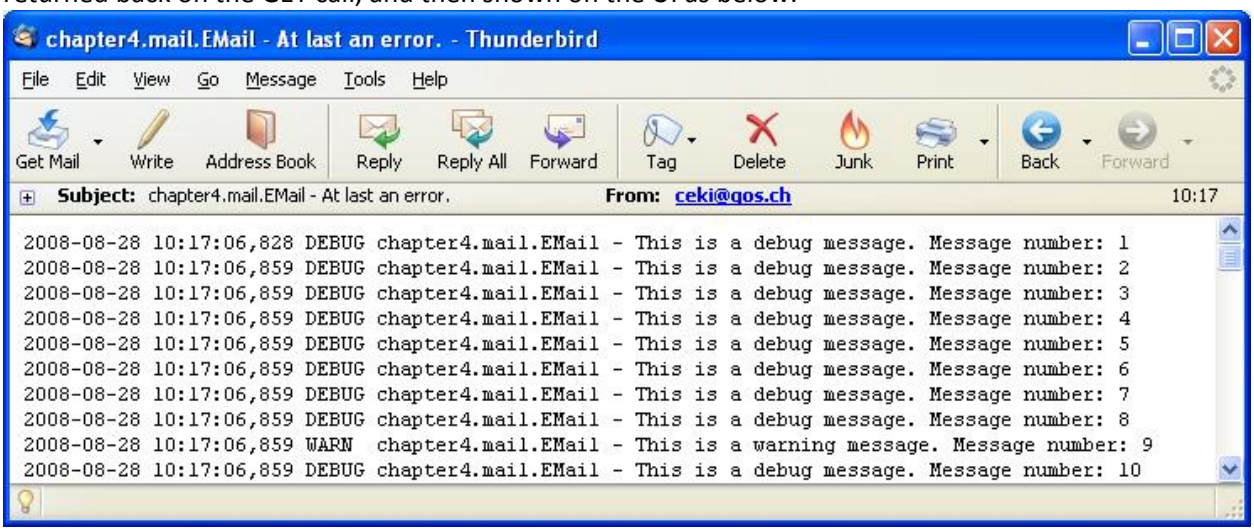
1. http://servername/getFromLog/:sender/:fromTime/:toTime
   This will return all logs for the particular sender for the given time range
2. http://servername/getFromLog/:sender/:fromTime/:toTime/:logging_level
   This will return only the mentioned level of loggings (users may be interested in looking at error level logs only)
3. http://servername/getFromLog/:sender/:fromTime/:toTime/:field_name/:field_value
   To return logs for the particular optional fields for the specified timeline

There should be no PUT and DELETE urls under normal circumstances, since these calls (definitely the POST) are expected to be made by other external systems to insert the logging. All calls should be authenticated, so no unknown system can put in its logging into the system.

There would be a separate admin credential and UI through which PUT and DELETE calls can be made to update or delete certain entries in exceptional circumstances.

Now that all the REST calls are setup, we can create a UI which can leverage the GET to show the logs on the UI. This page would display the JSON data returned by the GET in a more traditional format on the screen.

Attached below is a sample log file, whose data can be represented as JSON on the PUT call, and returned back on the GET call, and then shown on the UI as below:



Scenario 2: Expense Reports

In this scenario, you are tasked with making an expense reporting web application.

Users should be able to submit expenses, which are always of the same data structure: `id`, `user`, `isReimbursed`, `reimbursedBy`, `submittedOn`, `paidOn`, and `amount`.

When an expense is reimbursed you will generate a PDF and email it to the user who submitted the expense.

How would you store your expenses? What web server would you choose, and why? How would you handle the emails? How would you handle the PDF generation? How are you going to handle all the templating for the web application?

Since this involves PDF generation, we will need to use LaTeX or wkhtmltopdf. In either case the application would need to generate markup syntax (either its own for LaTeX, or HTML itlsef for wkhtmltopdf). Since the data structure is consistent, we can have a json object representing the same. When users submit expense, the front end would make a PUT request and pass the following JSON to the express server:

```
{
   "id":"123",
   "user":"user name",
   "isReimbursed":"false",
   "reimbursedBy ":"name of approver",
   "submittedOn":"submission date",
   "paidOn":"reimburse date",
   "amount":"200.05"
}
```

The server would store the data in MongoDB. When a reimbursement takes place, it is supposed to trigger a PDF generation event. So at this time, the nodeJS server would delegate the work to the worker code, which would generate the custom markup code based on the JSON data stored in MongoDB. This markup would then be either compiled to PDF by LaTeX or, converted to PDF by wkhtmltopdf. NodeJS package nodemailer would be used to send he emails.

When the markup is being generated, that would be done based on certain predefined PDF templates. These templates would be stored on the same file system, on which the nodeJS application is running. And the code can easily refer to the templates with a relative path. In this way, when a new template needs to be added, it can be simply put in the same folder as the others without the need to changing any code.

Scenario 3: A Twitter Streaming Safety Service

In this scenario, you are tasked with creating a service for your local Police Department that keeps track of Tweets within your area and scans for keywords to trigger an investigation.

This application comes with several parts:

- *An online website to CRUD combinations of keywords to add to your trigger. For example, it would alert when a tweet contains the words (`fight` **or** `drugs`) AND (`SmallTown USA HS` or `SMUHS`).*
- *An email alerting system to alert different officers depending on the contents of the Tweet, who tweeted it, etc.*
- *A text alert system to inform officers for critical triggers (triggers that meet a combination that is marked as extremely important to note).*
- *A historical database to view possible incidents (tweets that triggered an alert) and to mark its investigation status.*
- *A historical log of* all *tweets to retroactively search through.*
- *A streaming, online incident report. This would allow you to see tweets as they are parsed and see their threat level. This updates in real time.*
- *Long term storage of all the media used by any tweets in your area (pictures, snapshots of the URL, etc).*

Which Twitter API do you use? How would you build this so its expandable to beyond your local precinct? What would you do to make sure that this system is constantly stable? What would be your web server technology? What databases would you use for triggers? For the historical log of tweets? How would you handle the real time, streaming incident report? How would you handle storing all the media that you have to store as well? What web server technology would you use?

The **GET search/tweets REST api** can be used to search for certain keywords from tweeter. Details are provided by twitter here: https://dev.twitter.com/rest/reference/get/search/tweets. There will be a admin page where the search keywords can be configured. This can be customized as needed and hence can be expanded beyond the local search by using appropriate keywords.

Since this is a HTTP GET request which returns JSON from twitter, we are not bound by any specific technologies. The application would run on nodeJS on express servers and connect to twitter API using the HTTP GET url. The search events would be stored in a **relational database** (like Oracle/Postgre etc). An entry in the database would generate a database triggers which would then be used to generate emails and/or text messages.

We will use the **Nodemailer** module for nodeJS to generate emails (https://nodemailer.com/about/) and **TextMagic** module for nodeJS to send SMS alerts (https://www.textmagic.com/docs/api/node-js/).

The database would then be updated with the alert email/text information that was generated. These entries would also server as audit entries and can work as a historical database. There would be a worker module that connects to the database on one end and uses **websockets** to connect to a real time Incident Report. When a Database trigger is generated, it would also create an event on the worker thread to send a websocket message to the Incident Report page with the details, which will then display real time. The same data in a JSON format would be stored on **MongoDB** from where it can be used as a historical log to search through. The medial files from the feeds should be stored in **AWS or similar cloud locations**, with the file location stored in MongoDB.

Scenario 4: A Mildly Interesting Mobile Application

In this scenario, you are tasked with creating the web server side for a mobile application where people take pictures of mildly interesting things and upload them. The mobile application allows users to see mildly interesting pictures in their geographical location.

Users must have an account to use this service. Your backend will effectively amount to an API and a storage solution for CRUD users, CRUD 'interesting events', as well as an administrative dashboard for managing content.

How would you handle the geospatial nature of your data? How would you store images, both for long term, cheap storage and for short term, fast retrieval? What would you write your API in? What would be your database?

When a user makes a HTTP request the application will know the IP of the requester's public gateway (assuming the public gateway for the requester is in the same location). This IP can then be used to track down the geographical location of the requester. Any media via a POST will be stored along with the geographical location from where the request came. Hence when there is a request to GET pictures, from a user, only relevant pictures from that or nearby location will be used to render in the response. Details of geolocation providers can be found here: http://whatismyipaddress.com/geolocation-providers. This type of service can be used to geographically segregate the GET and POST requests and respond to them accordingly.

On a short term the images can be stored on the database for faster retrieval, though storing binary data on database could be costly. On a long term these images could be stored in the cloud like AWS, and the database could only hold the location of the file in the cloud. That way the long term storage is cheaper.

The API would be written in nodeJS, express and MongoDB. The administrative dashboard and the UI for the users would be written using jQuery and bootstrap. While the server side would be in nodeJS and MongoDB, which would interact with one of the geolocation providers and the AWS (or similar) cloud for image storage and retrieval. The reading and writing of images would be done to the database (or cloud) after being processed by ImageMagick or some similar image processing tool to make sure the images are rendered properly on most screens.