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?

Answer:

1) I would prefer to store the log entries as documents in MongoDB, as it is quick to lookup. Certain common field like date, timestamp and source followed by the customizable optional fields can be stored using a form with an option to create fields using unique key and value pairs.

2) Users can submit the log entries via a post request using an HTML form. The form must have an option for submitting additional fields in key/value format.

3) Users can query the log entries thus stored in the mongoDB by sending a GET request to the required endpoint. This can be done via an HTML form in a user-friendly manner in the system UI.

4) Log entries can be displayed in tabular form using HTML formatted by the CSS. Data about logs will be obtained from the MongoDB documents in JSON, which can then be rendered on the terminal using handlebars.

5) I would prefer to use Express.js to implement the server in a NodeJs environment, as it is the best web development framework that I am familiar with.

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?

Answer:

1) We can use MongoDB to store the expenses as documents. It would be simpler this time as the data has fixed pre-defined fields in this scenario.

2) I would use Express in NodeJs to create web server as it would fulfil the requirement and pairs well with MongoDB.

3) For emails, we can use Nodemailer, which is a NodeJs module used for sending emails. It can be used to create HTML as well as plain-text emails, add attachments and sent the emails with built-in SMTP support.

4) Javascript has many libraries to help generate pdf such as pdfmake, pdfKit and jsPDF. We will first use the json data from mongoDB with html code and css to create an html page, which will converted to a pdf using the pdfmaker library and emailed to the concerned user when reimbursement happens. We could also to integrate LaTeX with our application to generate pdf.

5) I would use handlebars to manage templating. I would keep one file *main.handlebars* as the template layout and use various other handlebars with it.

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.
• A 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?

Answer:

1) I would use the standard Twitter Search API, to find recent tweets(GET /2/tweets/search/recent) and use the “id” of the results in “statuses/show/:id to get “geo” and “place” field.

2) I will build a highly modular application where the place-related details is configured in a certain single location from where it can easily be modified to expand to other precincts with minimal hassles. All location specific data or code nuances should be captured in function call(from where it can be modified in case we require to port the application to different precincts) and should never be hardcoded. The application source code can be

organised as a generalized “template” or “skeleton” applicable to all precincts with function calls to handle the specificities. We can also opt for cloud storage so that the capacity can be easily stretched if required.

3) To make sure the system always remains stable, I would ensure a backup(shadow) application system is maintained by database and code replication at regular intervals to ensure smooth working in case of failure. Maintaining regular testing of the system, error logs and strong validation before deploying any change will be some of the practice to enhance stability.

4) I would use a Django web server in this case as its highly scalable and secure.

5) I would use influxdb for triggers as it is highly recommended for real time data like sensor data, or time series data.

6) For historical log of tweets, mongoDB would be the most appropriate as tweet data is unstructured and contains comments. This is suitably stored in mongoDB but can cause problem in some others databases like relational databases. Since comments can have a large range in size and number, we can store as subdocuments.

7) I would use django-celery-beat as a background job scheduler and task queue which would run jobs and triggers to notify when the incidents occur.

8) I would use Google cloud Platform(GCP) to store the large amounts of multimedia data, which would also have scalability advantages. GCP storage provides cost-optimizing options depending on how likely the stored files are to be accessed, for eg., we can use nearline or coldline storage in case the data is less often retrieved for low cost or we can choose standard storage for frequent access.

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?

Answer:

1) For the geospatial data we can use Google Location API, which has various useful endpoints for this application

2) I would store the images in Google cloud platform's storge(GCS) utility. For long-term cleap storage, I would avail the Nearline, Coldline or archival storage types in GCS and for short term, fast retrieval I will go for standard storage. For images that very frequently accessed, some caching tool redis will be employed.

3) We can use firebase authentication in NodeJs to write the API. We can also offer some RESTful API and also use ImageMagick to modify, edit and manipulate the images.

4) I would use a relational database like MySQL to store the data and links to the image resources stored in the Google cloud.