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?

I would utilize Node.js's console.log or logger function to log common and customisable fields as well as individual log entries since using built-in Node methods for logging is simpler when we use Node server to operate a website.

With MongoDB, I would **store** log entries as documents with some mandatory elements like timestamp, source, and content followed by the customisable fields. For CRUD tasks, MongoDB includes several really effective and simple data manipulation capabilities. using these functions users can **submit** and **query** the required logs.

I would let the user **see** their log entries on the server side log screen or console, which would be simpler to locate.

I would use **Node JS web server** framework as it is comaptible with MongoDb which were are using for logging.

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?

I would keep costs in a NoSQL database like **MongoDB** or **Couchbase**. These databases are useful for dealing with unstructured data, such as costs, which have a consistent data structure but may contain additional metadata or attachments. I'd create an expenditure collection and save each expense as a document with data for id, user, isReimbursed, reimbursedBy, submittedOn, paidOn, and amount.

I would chose **Server side Node.js** as my web server because of its interoperability with many javascript packages and its publicly available node package manager, Every time the user signs in, he must establish an account, and a session is generated using express-session.

I would utilize a third-party email service provider, such as **SendGrid** or **Mailgun**, to handle the emails. These services provide APIs for sending and tracking emails, which are essential for debugging and monitoring the application.

I'd use a library like **pdfkit** or **jsPDF** to produce PDFs. These packages enable you to generate PDFs programmatically while also modifying the layout and appearance. I'd create a PDF template and input it with data from the expenditure report.

The Web Application may be templated using **Bootstrap** or **Semantic UI**, both of which give simple functionality and a large number of ready-to-use templates for the specific case.

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?

We'd need to use **Twitter's Streaming API**, which delivers a continuous stream of real-time tweets. We may utilize the Tweepy Python package, which is a simple Twitter API wrapper.

To make the system extendable beyond the local precinct, we would need to design a scalable architecture that can handle bigger amounts of data. We might employ a cloud-based infrastructure like **AWS or Google Cloud**, which can be quickly scaled up or down as needed.

We would need to put up monitoring and recording systems to trace any issues that happen in order to ensure the system is always stable. To monitor system performance and logs, we may utilize technologies like **Elasticsearch and Kibana**.

We could utilize a current web framework like **Flask** or **Django** for the web server technology, which would allow us to simply develop a RESTful API for the service.

MongoDB may be used for triggers since it is a simple to use, extremely responsive NoSQL database capable of handling large volume transactions. **MongoDB** may also be used to store all media files and to keep a history database with all values and a flag for the threat level and investigation status, as well as a separate database to track past tweets and actions performed for searching.

SMS messages may be sent to selected police based on parsed tweets using the **Twilio** package, which can be utilized in Node.

We might utilize object storage, such as **AWS** or **Google Cloud Storage**, to store all of the material linked with the tweets. This would allow us to store and retrieve vast amounts of data with ease.

Web Server which will be used would be **Node** because it is the most reliable and compatible.

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?

To deal with the geospatial nature of the data, the user's latitude and longitude when completing the operation will be saved and may be projected back using **Google Maps Web API**, where the information can be presented at the level when compared.

For picture storage, we'd need a system that blends long-term, low-cost storage with short-term, quick retrieval. For long-term picture storage, we may utilize a cloud-based object storage service such as **Amazon S3** or **Google Cloud Storage**. We might utilize a content delivery network (CDN) like **Cloudflare** for short-term retrieval, which would store frequently visited photos at edge sites throughout the world for quick retrieval.

The API would be developed in client-side **Node**, which would include all required packages. When a user logs into a web application, Express and Express-session keep track of their session.

I would use **MongoDB** for the database because of its geographical capabilities. For storing user information and other non-geospatial data, we might alternatively utilize a relational database like PostgreSQL.