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 customizeable 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 can use a NoSQL database, such as MongoDB, as it will be a good choice for storing log entries. This is because NoSQL databases allow for flexible and scalable data modeling and can easily handle the different types of log entries that might be submitted. We can store log entries as documents in a collection in the database, with a schema that includes the common fields as well as any customizable fields.

The RESTful API can be used to allow users to submit log entries. The API will define endpoints for creating and updating log entries and will require authentication to ensure that only authorized users can submit log entries. Users can submit log entries in JSON format via the API.

Users can query log entries using a search textbox that allows them to specify the fields they want to search on. As the user enters the search, we can pass that search string as a query parameter in the API.

A webpage interface can be used to allow users to view and manage their log entries. The user interface can be built using a modern JavaScript framework such as React.

Finally, the web server can be implemented using a popular web framework we used i.e. Express.js. As this framework will provide endpoints for the RESTful API and serve the user interface to clients.

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?

To store the expenses, I will consider using a relational database management system such as PostgreSQL or MySQL, where each expense can be stored in a separate row in a table with columns for id, user, isReimbursed, reimbursedBy, submittedOn, paidOn, and amount. This will allow for efficient querying and sorting of the expenses based on any of these fields.

Then we can use a Python-based web framework such as Flask or Django as a web server. As frameworks have strong support for handling HTTP requests and responses, as well as templates for rendering HTML pages.

Moreover, to handle emails, we can use Python's built-in library called smtplib to send emails through a designated email server. The PDF generation can be handled by a python library called pdfkit, which provides a Python interface for generating PDFs.

Finally, for the templating of the web application, both Flask and Django have built-in templating engines that allow us to render HTML pages dynamically. With these templating engines, it will be possible to render expense forms for users to submit their expenses.

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?

To build this Twitter Streaming Safety Service, we can use the Twitter search stream API, which allows you to receive real-time streams of tweets based on certain keywords, locations, or users. But we will need to have access to Twitter's API and obtain authentication credentials to use it.

After that, to make this system expandable beyond your local precinct, we can add an administrative dashboard that allows other precincts to register for the service and configure their own set of keywords, locations, or users. For this dashboard, we can use AWS cloud service, and EC2 can be used to leverage virtual space.

To maintain stability, we can use AWS Elastic Load Balancer to distribute incoming requests from just one server to multiple if need be. We can use tools like AWS Prometheus to monitor the system's health and performance and implement automatic alerting when issues arise.

For the web server technology, we can use a modern server-side framework like Node.js with Express, with Python. These frameworks provide many features out of the box that can be useful for building a web application, such as routing, middleware, handlebars, and more.

For the databases, we can use NoSQL databases such as DynamoDB to store the historical log of tweets and AWS S3(Simple Storage Service) to store the media used by the tweets. We can use Redis to cache frequently accessed data like the incident report or the keywords for triggering alerts.

For handling the real-time streaming of tweets and the incident report, we can use a tool like Socket.IO, which allows you to build real-time applications using WebSockets. This will allow you to push updates to the client as new tweets are received and processed.

For storing the media used by the tweets, as we discussed we can use a cloud-based storage service like AWS S3 or Google Cloud Storage, which provides scalable and durable storage for objects like images, videos, and other media.

Finally, for the web server technology, we can use a modern server-side framework like Node.js with Express, or Redis, with Python. These frameworks provide many features out of the box that can be useful for building a web application, such as routing, middleware, handlebars, and more.

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 handle the geospatial nature of the data, we can use a database that supports geospatial indexing, such as MongoDB or PostGIS. This will allow us to store location data along with each interesting event and perform geospatial queries, like finding events within a certain radius of a user's location.

For storing images, we can use a combination of long-term, cheap storage and short-term, fast retrieval solutions, more likely Amazon S3 or Google Cloud Storage. Or we may also use a content delivery network (CDN) like Cloudflare to ensure fast retrieval of images by users.

For the API, we can use a modern web framework like Node.js with Express or Ruby on Rails. This will allow me to quickly build a robust API that can handle user authentication, CRUD operations for interesting events and users, and other administrative features.

For the database, we will consider using MongoDB. MongoDB will be a good option if we need to perform complex queries on geospatial data.