Lab 9

As part of the project, many teams decided to write their own backends, so there was demand to cover ways of using JavaScript in the backend.

There are multiple ways of using JavaScript in writing server-side code, a few of which are:

- Standalone Node.js server
- Hooks/endpoints as part of the frontend framework
- Serverless functions (such as using Firebase/Supabase)
- Edge functions (using Netlify, CloudFlare Workers, Vercel, ...)

Standalone server

This is by far the most common and traditional method of writing server-side code: actually having a *server* 😁. It works just like servers written in other languages such as Python or Java. Orchestrated with frontend architecture, this can be a separate server acting just as an API (coupled with a separate backend for hosting your app), or it can be the server that actually hosts your frontend application (perhaps even performing SSR).

While plain Node.js (https://nodejs.org/en/) alone can be used to write a web server, it can become quite cumbersome rather quickly, and so people prefer to use libraries. The most popular option is Express (https://expressjs.com/).

The simplest ("Hello World") Express application you can write looks something like this:

```
const express = require('express');

const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`);
});
```

So first we have the import (in CommonJS style) followed by instantiating a server instance. Then, we declare the routes and HTTP methods (https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods) our server supports; for example, we can see above that our app can handle `GET` requests to the `/` path only.

We define a callback function that has access to the request and response objects; we get information about the request from the former and use the latter to send the response (adding headers and other information if needed).
Lastly, we call `app.listen` which causes our program to stay put instead of exiting, waiting for incoming connections to the server (as handled by Node's event handling mechanism, similar to the one in the browser).

You can handle different methods to different paths:

```
app.post('/users', (req, res) => {
  // Create a user in the db after validating req.body
});
```

or tell it to treat a folder as having static content, in which case it will be used whenever a `GET` request doesn't match any of the registered handlers:

```
app.use(express.static('public'))
```

Find more examples and references in the official Express documentation (https://expressjs.com/)

There are alternatives to Express that you can explore if you want, such as NestJS (https://nestjs.com/) or Fastify (https://www.fastify.io/) and many others.

Serverless Functions

A **Serverless Function** is a single-purpose function hosted on infrastructure maintained by Cloud companies. In other words, it's a feature of Serverless platform providers such as Firebase and Supabase that allows you to write some custom logic that runs securely inside their environment without worrying about the boilerplate and complexities associated with running your own full-fledged server.

In particular, let us consider Firebase Cloud Functions (https://firebase.google.com/docs/functions). Firebase allows you to write Functions in a wide variety of languages (including JavaScript and Python) that run as a result of some event happening. Events that trigger functions include operations on Firestore, authentication (ex: user signing up), operations on Cloud Storage (ex: uploading a new file), a scheduled run (like a cron job), or even simply an HTTP call to the function directly.
The function can then respond to the event by performing any task such optimizing images uploaded to the storage, notifying users when they get a follower, deleting inactive accounts, sanitize data in the database, and much more (https://firebase.google.com/docs/functions/use-cases).

For example, this is a function that runs whenever a user deletes their account, updating Firestore accordingly:

```
export const processAccountDeletion = functions.auth
  .user()
  .onDelete(async ({ uid }) => {
    const db = getFirestore();

    db.doc(`users/${uid}`).delete();
    // Update statistics - decrement user count
    await db.doc("misc/stats").update({ users: FieldValue.increment(-1) });
  });
```

And this is an example of another function that cascades the deletion of a document (representing a story entity), deleting its subcollections and related entities as well:

```
export const deleteStory = functions
    .region("europe-central2")
    .https.onCall(async (storyId: string, context) => {
      const storyRef = storyDoc(storyId);
      const story = await storyRef.get();
      if (context.auth?.uid !== story.data()?.metaData.authorId) {
        throw new functions.https.HttpsError("permission-denied", "Only the aut
      }
      const translations = await storyRef.collection("translations").get();
      for (const translation of translations.docs) {
        // ... Delete the translation and its assets (subcollections) ...
      }
      const likes = await db.collectionGroup("likes").where("storyID", "==", st
      await Promise.all(likes.docs.map((like) => like.ref.delete()));

      await storyRef.delete();
    });
```

You can find more examples in the official Functions Sample Library (https://github.com/firebase/functions-samples).

Hooks & Endpoints in SvelteKit

In SvelteKit, you can write **Hooks** (https://kit.svelte.dev/docs/hooks) that run on the server-side before handling a user request. They are similar to middleware (https://en.wikipedia.org/wiki/Middleware) in other frameworks.

To write hooks in SvelteKit, simply export a `handle` function from `src/hooks.js`:

```
/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
  if (event.url.pathname.startsWith('/custom')) {
    return new Response('custom response');
  }

  const response = await resolve(event);
  return response;
}
```

This function will intercept every request (whether during the app is running or while prerendering) and determines the response, in addition to possibly modifying response headers or body, or more.

If this is more flexible than you need and you just want a request handler for a particular endpoint, you can use exactly that: **Endpoints** (https://kit.svelte.dev/docs/routing#endpoints). Endpoints are routes written in JS/TS that allow you to handle requests to the specified URL and HTTP method(s).

Endpoints can be **standalone** or **page** endpoints. Standalone endpoints are independent of pages and act just like Express endpoints. They are usually used to implement some sort of API, and can look like so:

```
// src/routes/items/[id].js
import db from '$lib/database';

/** @type {import('./__types/[id]').RequestHandler} */
export async function get({ params }) {
  // `params.id` comes from [id].js
  const item = await db.get(params.id);
  if (item) {
    return {
      status: 200,
      body: { item }
    };
  }
  return { status: 404 };
}
```

Page endpoints are similar to standalone ones except that they have the same file name as a page (that ends in the `.svelte` extension), and whatever you return in the `body` becomes available to the page's Svelte component as props.

Note that endpoints may not be compatible with prerendering (and hence `adapter-static`) since static files cannot handle requests other than `GET`. When prerendering, SvelteKit will run your hooks and endpoints at compile time only.

Edge Functions

Edge Functions (https://www.netlify.com/blog/edge-functions-explained/) are the new kid on the block. Similar to Serverless Functions, they allow you to write simple server-side functions without worrying about the complexities of setting up servers, but the main difference is that they run on “the edge”. This makes sense only in Content Delivery Network (CDN) services, where “the edge” simply means the server geographically closest to the user. In Firebase, on the other, you deploy your Function in exactly one geographical location (ex: `us-central-1`). Other than that, they generally provide the same features

The most popular CDN providers that support Edge Functions as of today are Netlify (https://docs.netlify.com/netlify-labs/experimental-features/edge-functions/), Vercel (https://vercel.com/docs/concepts/functions/edge-functions), and CloudFlare (https://workers.cloudflare.com/). Each provider will have its own way of defining Functions, but the easiest way to use them is by defining hooks and endpoints in SvelteKit as usual, and then using the respective adapter of your target platform, which will automatically transform them to Edge Functions of that platform.