Lab 6

Today, we're going to apply all the strategies covered in the lecture by creating a SvelteKit application, testing out the different rendering strategies, adding Firebase to our project, and deploying it to GitHub Pages using GitHub Actions.

First, let's start by creating a project from template, following the docs (https://kit.svelte.dev/docs/):

```
npm create svelte sample-app
```

We want TypeScript support, and let's select all the quality assurance tools just to explore them (we will use them later).
This will create a demo "Sverdle" app that we can run using `npm run dev`. Follow the steps in the console to install dependencies and create a local git repo.

The first thing you may notice is the new `svelte.config.js` file which contains all the configuration needed for SvelteKit. The most important configuration option in this file is the `adapter` (https://kit.svelte.dev/docs/adapters), which is a plugin that generates the final output to be deployed on your preferred target. We will try out different adapters to see the different rendering strategies.

This config file is used by the SvelteKit plugin used in `vite.config.js`.

Rendering strategies

By default, SvelteKit uses `adapter-auto` which tries to detect the environment and pick another appropriate adapter (CloudFlare, Netlify, or Vercel). All adapters have the same behavior during development, but they output different code when building for production.

Server Side Rendering

To perform server-side rendering, we naturally need a server. The most suitable one to use here would be Node.js with the official `@sveltejs/adapter-node` (https://github.com/sveltejs/kit/tree/master/packages/adapter-node). Let's remove the `auto` adapter and use this one instead.

```
npm uninstall @sveltejs/adapter-auto
npm install --save-dev @sveltejs/adapter-node
```

And replace the import in the `svelte.config.js`.

Now, let's go back to the browser and take a look at the page source. You will see that it comes with all initial HTML already there, including useful meta tags in the head. This helps search engines index our website better and the user can see the content even before it becomes interactive.

It's important to note that this version of the app depends on the existence of a Node.js server. It doesn't generate static HTML/JS files that we can just put on any static site hosting solution (like GitHub Pages).

Check out the `build` folder after running `npm run build` to see how it works.

Static Site Generation

On the other hand, if we know that we can serve the same exact page to all users, we can generate all of the HTML pages beforehand and just serve them using any static file server. Note that the JavaScript can change some of the content when it arrives at the user's side (e.g.: to fill in the user's information) using hydration, but SSG would still work. This is comparable to what we were doing before using SvelteKit, when we had static HTML files and just included the built JS file in it.

So, let's simply replace the node adapter above with `@sveltejs/adapter-static` (https://github.com/sveltejs/kit/tree/master/packages/adapter-static):

```
npm uninstall @sveltejs/adapter-node
npm install --save-dev @sveltejs/adapter-static
```

and then replace the import in `svelte.config.js` and add `export const prerender = true;` in `+layout.ts`. We also need to remove the Form Actions (https://kit.svelte.dev/docs/form-actions) from `/sverdle` because it uses some SvelteKit functionality that is not compatible with prerendering.
One more thing to change is to set `export const trailingSlash = 'always';` in the same file (`+layout.ts`) to tell SvelteKit to generate all our pages as `route-name/index.html` so we wouldn't have to add `.html` to all of our links and maintain consistent behavior with other adapters.

Now, all that's left is to `npm run build` (and take a look at the `build` directory), and then run any kind of server inside the `build` directory or deploy to some hosting service (for example: `python -m http.server`).

Client Side Rendering

To perform pure client-side rendering (which is highly unadvisable), simply turn off the `prerender` and `ssr` options by exporting variables with these names from `+layout.ts`.
Notice how we can use different rendering strategies for different parts of our application?

Deploying via CI/CD

First things first, we need to create a GitHub repo, but you already know how to do that; just don't initialize it with any content (README or LICENSE or so) to avoid having commit conflicts. If you haven't already created a local repo, do so using `git init` and then follow the instructions from GitHub to push it.

There are so many CI/CD solutions out there (such as CircleCI, Jenkins, Travis CI, …), but the one we'll go with today is GitHub Actions (https://docs.github.com/en/actions).
To write our GitHub Action, we need to create a YAML file (with any name) under the `.github/workflows` folder.

```
name: Build and Deploy

on:
  push:
    branches:
      - master

permissions:
  contents: write

jobs:
  build-and-deploy:
    name: Build and Deploy to GH Pages
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: Setup Node.js v16
        uses: actions/setup-node@v3
        with:
          node-version: '16'
      - name: Install dependencies
        run: npm ci
      - name: Build website
        run: npm run build
      - name: Deploy to GitHub Page
        uses: JamesIves/github-pages-deploy-action@v4
        with:
          folder: build
```

To find out what exactly you can write in this YAML file, refer to the Workflow Syntax (https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions). Shortly, there are 3 required top-level keys: `name` , `on` , and `jobs` . Name is obvious, `on` defines the events that will trigger your workflow, and `jobs` defines the pipelines that will run as part of the workflow. The `permissions` section tells GitHub that we allow this Action to write to our repo, which is needed by the last step to deploy to the `gh-pages` branch.

You must have at least one `job`, which consists of many steps. Each `step` must at least have either the `run` or `uses` key, but not both. It's recommended to also give a `name` to all steps for better readability in GitHub's logs.

Don't forget to enable GitHub Pages publishing from the `gh-pages` branch in your repo settings. It may take a couple of minutes for the change to kick in.

An important thing to fix when using SvelteKit with GitHub Pages (https://kit.svelte.dev/docs/adapter-static#github-pages) is to disable Jekyll rendering (enabled by default) so that it doesn't ignore the generated `_app` directory. We also need to adjust the base path (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base) in case we're using a normal repo, not the one with the user site (`username.github.io`).

The first one is simple: just add an empty file called `.nojekyll` inside the `static` folder.

To fix the base URL, adjust your `svelte.config.js` as follows:

```
const dev = process.argv.includes('dev');

const config = {
  kit: {
    // ...
    paths: {
      base: dev ? '' : '/your-repo-name',
    },
  },
};
```

But for this to work properly, you need to use relative URLs everywhere in your app, and they will all become relative to the `base` you provide. From the relevant page in the docs (https://kit.svelte.dev/docs/configuration#paths), this is how to do it:

```
<script lang="ts">
  import { base } from '$app/paths';
</script>

<a href="{base}/your-page">Link</a>
```

Firebase

Finally, the last missing component to develop a modern website is a server. Firebase provides us with tons of services with a very easy to use API that we do not need to write our own server.

Start by navigating to the Firebase console (https://console.firebase.google.com/) and create a new project. Then, after it's created, you need to register a web app and copy the code snippet provided into the script section of `+layout.svelte` (or directly to `+layout.ts`). Don't forget to also `npm install firebase`.

This initialization object (which contains an API key) can be safely shipped to the client since the security will be done on the dashboard using Security Rules (https://firebase.google.com/docs/rules).

Authentication

Go back to Firebase console, find the Authentication service, and enable logging in with email/password. Now add a login form with bindings to the `email` and `password` variables (you should already know how to do that), and just add the following snippet to the component:

```
import { signInWithEmailAndPassword, getAuth } from 'firebase/auth';

async function login() {
  const auth = getAuth();
  const { user } = await signInWithEmailAndPassword(auth, email, password);
  console.log(user);
}
```

Or to register, use the `createUserWithEmailAndPassword` instead.
And that's it! You now have user authentication in your app!

> Note: by default, logging in with an OAuth provider (e.g.: Google) requires your app to be running under an authorized domain (configurable in the dashboard through `Authentication > Settings > Authorized domains`). For local development, the easiest way is to change the port to 80 by adding `server: { port: 80 }` to your `vite.config.ts`.

Database

Firestore (https://firebase.google.com/docs/firestore) is a NoSQL database with real-time updates that is also part of Firebase services. Find it in the dashboard and enable it in test mode (until you read about Security Rules and use them). The data is structured in terms of collections of documents, where a document is simple a JSON object.

All of the relevant imports will be from the `firebase/firestore` module.
To start with, this is how to get the data of a particular document:

```
const db = getFirestore();
const reference = doc(db, 'user/user1234');
const snapshot = await getDoc(reference);
const data = snapshot.data();
```

Adding a document to a collection is just as simple:

```
const todos = collection(db, 'todos');
const newRef = await addDoc(todos, {
    done: false,
    text: 'Example To Do',
});
console.log(newRef.id); // Get the document ID
```

And finally, to listen for any updates and not just get the data once, you can use `onSnapshot`:

```
const ref = collection(db, 'todos');
onSnapshot(ref, (snapshot) => {
    todos = snapshot.docs.map(doc => doc.data()) as Todo[];
});
```

This will call it once in the beginning to get the initial data, and then again any time the data in the given reference gets updated (created/edited/deleted). This works for references to whole collections, single documents, or even collection queries.

Homework

Take the website from the previous assignment (with Svelte components) and adapt it to use SvelteKit. The simplest way is to just create a skeleton SvelteKit project from template and copy over your components and pages to it. You should have at least 2-3 routes with navigation links between them.

Write GitHub Actions to build and deploy your site to GitHub Pages when you push to the main branch.

Your git repo should not contain any generated files (such as `build`, `node_modules`, `.svelte-kit`, …)