Lab 7

It’s time to learn another, way more popular, frontend framework: React (https://reactjs.org/)!

How to use React

Rather than just repeating the lecture content about React, it might be more helpful to compare it to the framework we already know and practiced - Svelte. To do so, check out the Component party (https://component-party.dev/) to compare simple examples of how to do various stuff in both frameworks, and also take a look at the illright/react-for-svelte-devs (https://github.com/illright/react-for-svelte-devs) GitHub repo to see how the various Svelte concepts can be mapped to React.

It is also recommended to go through the Getting Started (https://reactjs.org/docs/getting-started.html) page of React’s docs.

In a nutshell, there are a few ways of creating React projects:

- Playgrounds: CodePen (https://codepen.io/topic/react), CodeSandbox (https://codesandbox.io/s/new) (try writing react.new (https://react.new) in your browser), StackBlitz (https://stackblitz.com/fork/react)
- Add script directly to HTML file
- Create-React-App
- Next.js (or any other meta-framework)

Playgrounds

Playgrounds are like online IDEs that come preconfigured with many templates for the various stacks you might want to bootstrap your app with. They can be useful for starting out a project in an easily disposable environment, but as your project grows and needs custom workflows, other dependencies, collaboration, etc., you will most likely need to migrate to a proper development environment with a more complicated version control system.

There are many online code playgrounds out there, but the most popular ones are CodePen (https://codepen.io/), CodeSandbox (https://codesandbox.io/), and StackBlitz (https://stackblitz.com/).

In real-life projects you will most likely not use a playground, so we’re jumping straight to a real workflow.

HTML file

Since React is just a library that exposes some interface, we can include it as a regular script as such:

```
<script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin><
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" cross
```

and then you'll have access to functions such as `React.createElement` and `ReactDOM.createRoot`, allowing you to make use of all of React's features (except for JSX) but in a less pleasant format than if you use dedicated JavaScript modules.

It is also possible to use JSX in the HTML file by loading Babel and using `<script type="text/babel"> /* JSX here */</script>` as outlined in the docs (https://reactjs.org/docs/add-react-to-a-website.html), but that's not recommended as it slows down your website and makes it unsuitable for production (but then again, if you're not using a bundler, how *suitable for production* is your website to begin with?).

CRA

Of the many ways described in the docs to create a new React app (https://reactjs.org/docs/create-a-new-react-app.html), Create-React-App (https://create-react-app.dev/) (or CRA for short) is the most popular way of bootstrapping a React project (if you're not using SSR). Shortly speaking, it creates a project with lots of sensible configuration defaults and tools that help you get productive right away. For example, it sets up Webpack (https://webpack.js.org/) as the bundler with JSX and TypeScript support, ESLint as the linter, CSS autoprefixer, a unit tests runner, a local development server, and much more. The only downside is that you cannot customize the configuration of any of the above.

To get started, simply run `npx create-react-app my-app` (using NPM v5.2+) or `npm create react-app my-app` (using NPM v6+), which will create a folder called `my-app` which has a `public` folder including all static assets to be copied as they are, and a `src` folder which will include your React components.

Adding TS support

To create a React project with TypeScript support from scratch, just add the `--template` parameter to the creation command: `npm create react-app my-app -- --template typescript`.

If you already have an existing project and need to add TypeScript to it, follow the instructions in the CRA docs (adding TS) (https://create-react-app.dev/docs/adding-typescript/).

Removing auto-config

If, for whatever reason, you're not satisfied with the defaults enforced by Create-React-App and would like to have full control (there is no middleground), then you should simply run `npm run eject` and it will replace `react-scripts` with ALL of its dependencies that were hidden from you, in addition to bringing out all of the configuration files that it was using internally to the `config` directory and the scripts to the `scripts` directory. Be careful, these files can be rather huge.

Next.js

Next.js is the equivalent of SvelteKit for React. It makes it very simple to write a production-ready website with SSR support. And instead of repeating everything on their website, I recommend you simply go follow the guide there: https://nextjs.org/learn/basics/create-nextjs-app (https://nextjs.org/learn/basics/create-nextjs-app)

Exercise

Since we already learned and practiced one framework (Svelte), it would make it easier to comprehend the new concepts and compare the frameworks if we build the same app from last time (https://hackmd.io/@aabounegm/FE-lab4), but in React this time.

As a refresher, we had a simple chat room application built using `Socket.IO`. We had 2 components: `Message` and `Input`, and combined them in the main file. We will not go through the dependency installation this time or explaining the responsibility of every bit since that was done last lab.

Start by creating a React project using CRA's TypeScript template. We don't need to configure anything else, so we can jump straight to the code. Add the following interface to **`src/message.ts`**:

```
export interface Message {
  username: string;
  message: string;
}
```

Then, we need a folder for each component with 2 files, one for the JSX and the other for the CSS. Let's start with the stateless component for displaying a message:

`src/message/Message.tsx`:

```
import type { FC } from 'react';
import classes from './Message.module.css';
import { Message as MessageType } from '../message';

const Message: FC<MessageType> = ({ message, username }) => {
  return (
    <li className={classes.li}>
      <span style={{ fontWeight: 'bold' }}>{username}: </span>
      {message}
    </li>
  );
};

export default Message;
```

with the following in **`src/message/Message.module.css`**:

```
.li {
  padding: 0.5rem 1rem;
}
.li:nth-child(odd) {
  background-color: #f8f8f8;
}
```

Next, we move on to the input component.

`src/input/Input.tsx`:

```
import { type FC, type FormEvent, useState } from 'react';
import classes from './Input.module.css';
import type { Message } from '../message';

interface InputProps {
  onSend: (message: Message) => void;
}

const Input: FC<InputProps> = ({ onSend }) => {
  const [username, setUsername] = useState('');
  const [message, setMessage] = useState('');

  function send(e: FormEvent) {
    e.preventDefault();
    if (!username || !message) {
      alert('Please enter a username and a message');
      return;
    }

    onSend({
      message,
      username,
    });
    setMessage('');
  }

  return (
    <form onSubmit={send} action="" className={classes.form}>
      <input
        className={classes.input}
        autoComplete="off"
        value={username}
        onChange={(e) => setUsername(e.currentTarget.value)}
        placeholder="Name"
        style={{ flexGrow: 0 }}
      />
      <input
        className={classes.input}
        autoComplete="off"
        value={message}
        onChange={(e) => setMessage(e.currentTarget.value)}
        placeholder="Type your message..."
      />
      <button>Send</button>
    </form>
  );
};

export default Input;
```

I'll leave it as an exercise to you to figure out how the rest translates from a Svelte SFC, and how to modify the CSS to work with Modules (since React does not do scoping automatically).

Lastly, we need to put it all together in our main entry point.

`src/App.tsx`:

```
import { useState } from 'react';
import Message from './message/Message';
import Input from './input/Input';
import { Message as MessageType } from './message';

function App() {
  const [messages, setMessages] = useState<MessageType[]>([]);

  function onSend(message: MessageType) {
    setMessages((prevMessages) => [...prevMessages, message]);
  }

  return (
    <main>
      <ul>
        {messages.map((message) => (
          <Message {...message} key={message.username + message.message} />
        ))}
      </ul>
      <Input onSend={onSend} />
    </main>
  );
}

export default App;
```

Notice that we skipped the whole `Socket.IO` connection part for brevity, but feel free to add it back up yourself. The server will be left up and running until the end of the semester.

Homework

To crush it on your midterms :)

If you feel bored, just continue the remaining parts of the lab exercise 😉