THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

COMPSCI 732

SOFTENG 750

Creating and Consuming APIs

Agenda

- Crash-course in Node.js / Express
  - express, express-router, body-parser
  - My first API
  - Organizing backend code
- Debugging vs production with create-react-app frontend and express backend
- Consuming APIs
  - fetch()
  - axios
- Integration with React

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Three

Writing a backend with Node.js / Express

Node.js / Express

- **Node.js:** Allows the creation of JavaScript apps outside the browser
  - Webpack / Babel / etc. run in this environment
- **Express:** A node package allowing developers to easily write backends
  - Developers create server-side routes, which perform various actions based on the path / URL, HTTP method (e.g. GET, POST), and various other parameters
  - Very pluggable – many packages add-on and provide additional functionality
  - Install using npm:

```
npm install express
```

The Simplest Express Server™


```
import express from 'express';

const app = express();
const port = process.env.PORT || 3000;

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

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

When a client makes a `GET` request…
To "/" (e.g. `http:/localhost:3000/`)…
The given function will be called, which sends the text "Hello World!" back to the client.

Starts the server running on the given port. When the server is up and running, the given function is called, which will print a message to the console.

Serving JSON

- JSON is the modern data interchange format of the web
  - JavaScript seamlessly supports JS Object ←→ JSON conversion
- To serve JSON from Express apps, we simply use `res.json()` as shown here:

```
const todos = [
   { text: 'Do stuff', completed: false },
   …
];
…
app.get('/todos', (req, res) => {

   res.json(todos);
});
```

Serving static files

- In addition to running our own code when we receive a request, we may just want to serve static files within our project.
  - For example, all the client-side files that are served by Webpack on our dev server…
- To do this in Express:

```
import express from 'express';
import path from 'path';

const app = express();
…
app.use(express.static(path.join(__dirname, 'public')));
```

Will serve all content in the `public` directory.

E.g. if there's a file called `image.png` in `./public`, we can access it using `http://localhost:3000/image.png`

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Three

Express Routing and code organization

Express Router & code organization

- Rather than stick all our Express routes in one file, we often want to separate them in some way
- We can use Express Router for this purpose:
  1. Define a Router in a separate file
  2. Configure route handlers for that router as shown in previous slides
  3. Import and "use" that router from main (or any other) file
- Many different ways we could organize our routes
  - The following slides show *one* way – but you're free to organize how you like, as long as it's understandable to you and the markers ☺

Express Router & code organization


```
const app = express();

app.get('/foo', (req, res) => { … });

const mainRouter = express.Router();
app.use('/main', mainRouter);

mainRouter.get('/bar', (req, res) => { … });

const childRouter = express.Router();
mainRouter.use('/child', childRouter);

childRouter.get('/baz', (req, res) => { … });
```

1)

2)

3)

Quiz: Assume the webapp shown here is listening on http:/localhost:3000/.

What URL would result in each of the three highlighted route handlers being called?

Express Router & code organization


```
const app = express();

app.get('/foo', (req, res) => { … });

const mainRouter = express.Router();
app.use('/main', mainRouter);

mainRouter.get('/bar', (req, res) => { … });

const childRouter = express.Router();
mainRouter.use('/child', childRouter);

childRouter.get('/baz', (req, res) => { … });
```

1)

Quiz: Assume the webapp shown here is listening on http:/localhost:3000/.

What URL would result in each of the three highlighted route handlers being called?

1. http://localhost:3000/foo

Express Router & code organization

```
const app = express();

app.get('/foo', (req, res) => { … });

const mainRouter = express.Router();
app.use('/main', mainRouter);

mainRouter.get('/bar', (req, res) => { … });

const childRouter = express.Router();
mainRouter.use('/child', childRouter);

childRouter.get('/baz', (req, res) => { … });
```

2)

Quiz: Assume the webapp shown here is listening on http:/localhost:3000/.

What URL would result in each of the three highlighted route handlers being called?

1. http://localhost:3000/foo
2. http://localhost:3000/main/bar

Express Router & code organization

```
const app = express();

app.get('/foo', (req, res) => { … });

const mainRouter = express.Router();
app.use('/main', mainRouter);

mainRouter.get('/bar', (req, res) => { … });

const childRouter = express.Router();
mainRouter.use('/child', childRouter);

childRouter.get('/baz', (req, res) => { … });
```

3)

Quiz: Assume the webapp shown here is listening on http:/localhost:3000/.

What URL would result in each of the three highlighted route handlers being called?

1. http://localhost:3000/foo
2. http://localhost:3000/main/bar
3. http://localhost:3000/main/child/baz

Express Router & code organization

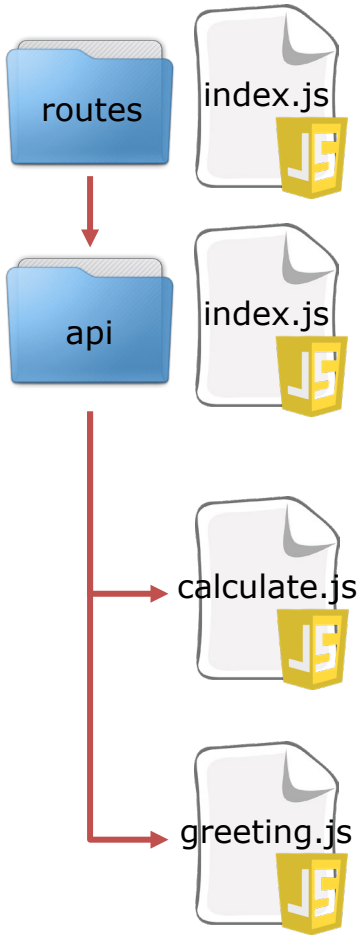
Route	File / Folder structure	Code

/	routes / index.js	`import api from './api';`
		`router.use('/api', api);`
		`export default router;`

/api	api / index.js	`import calculate from './calculate';`
		`router.use('/calculate', calculate);`
		`import greeting from './greeting';`
		`router.use('/greeting', greeting);`
		`export default router;`

/api/calculate	calculate.js	`const router = express.Router();`
		`router.post('/', (req, res) => {`
		`  …`
		`});`
		`export default router;`

/api/greeting	greeting.js	`const router = express.Router();`
		`router.get('/', (req, res) => {`
		`  …`
		`});`
		`export default router;`



/api is a **folder**. If we **import** a folder, JavaScript will actually `import` a file named **index.js** *inside* that folder.

Instead of importing /api/index.js, we can import /api as shorthand.

Then, from our main file:

```
const app = express();
…
import routes from './routes';
app.use('/', routes);
```

… And much, much more!

- Express is a simple yet comprehensive framework for building APIs / REST services / other backend logic. Much of it is out of the scope for this course
  - … but you may need more info for your projects!
  - Check out the Express.js website.

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Three

Consuming APIs – fetch()

fetch()

- `fetch()` is the modern JavaScript way to send HTTP requests & receive responses (i.e. to call our APIs)
  - Supported in browsers & server environments
- Sending a GET request is very simple:

```
fetch("/api/greeting")
    .then(response => response.json())
    .then(json => spanGreeting.innerHTML = json.message);
```

Send a GET request to /api/greeting

When the request completes, convert the response to a JS object

When the conversion is complete, display the received message *(this is standard JavaScript, NOT React!!)*

- Can also use `async / await` if desired

fetch()

- Sending a POST (or other type of HTTP message) is slightly more complex:

```
fetch("/api/calculate", {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify({
        a: parseInt(txtA.value),
        b: parseInt(txtB.value)
    })
})

    .then(response => response.json())

    .then(json => spanResult.innerHTML = json.result);
```

Send a request to /api/calculate…

This is a POST request

The Content-Type of the request body is JSON

The request body is a JSON object with two properties: a and b

Parse the result as JSON and display it

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Three

Consuming APIs – axios

Axios

- A library providing HTTP connectivity
- More comprehensive suite of functionality compared to fetch()
- But has a similarly simple programming model
- Consistency when calling axios functions from both frontend and backend JavaScript code
- Also promise-based, like fetch()

Axios - installation

- To install in your node.js or React apps:

```
npm install axios
```

- To install in your plain HTML/CSS/JS websites:

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

Axios – usage example

First argument: URL to target

```
axios.get("/api/greeting")
    .then(response => spanGreeting.innerHTML = response.data.message);
```

Function names equal to HTTP method names (e.g. GET, POST)

```
const body = {
    a: parseInt(txtA.value),
    b: parseInt(txtB.value)
};

axios.post("/api/calculate", body)
    .then(response => spanResult.innerHTML = response.data.result);
```

(optional) Second argument: HTTP request body (will be converted to a JSON string by default)

response object: contains the status code / text, headers, response body, and other useful info. The response body is parsed as JSON and converted to a JS object, by default.

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Three

React apps with a Node.js / Express backend

Backends for React webapps

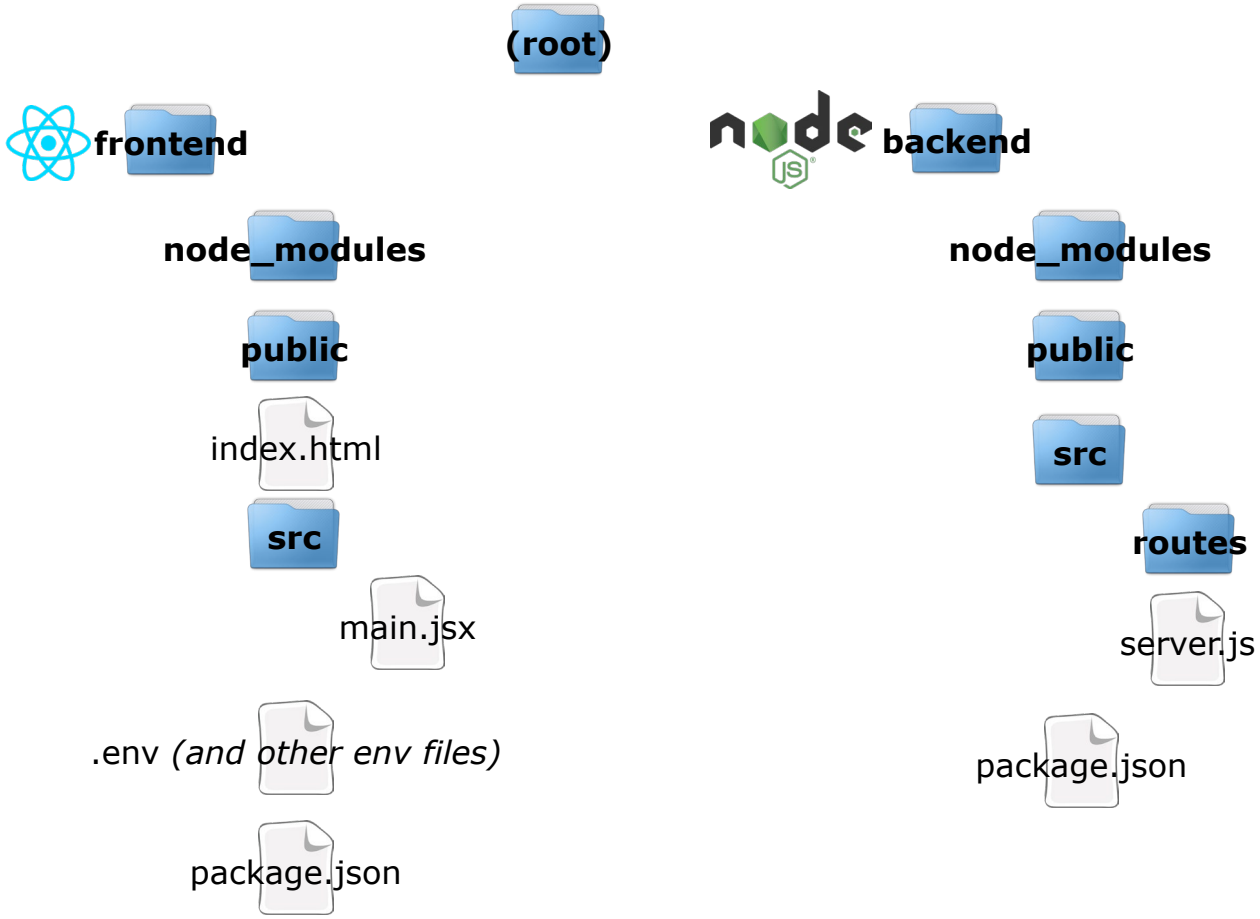
- React doesn't require the use of any particular backend. Its only requirements:
  - Can serve static files
  - Appropriately serves "index.html" on a GET request to any valid application URL (for proper client/server-side routing)
- Express can certainly do this – but so can any other backend framework you may already know
  - React apps created with Vite keep React's backend-agnostic approach
  - React apps created with other toolchains may make assumptions regarding backend (e.g. Next.js API functions are node-based)

Development with Vite frontend, Express backend

- During development, Vite runs its own server to support hot module reloading, etc.
- Good idea to use an **environment variable** (usually through an *env file*) to point your frontend to your backend
  - That way, in development we can have our Vite server and Express server running at the same time, and point one towards the other
  - And, in production, we can deploy the two in different (or the same) places easily

Vite + Express – Possible project layout

- One possible project layout to organize your source code:



This layout will keep the frontend and backend `node_modules` folders separate, which is ideal – usually the client and server depend on mostly different packages.

Use of env files

- Example:
  - Your dev environment backend is http://localhost:3000
  - You'll be deploying your backend to https://www.foo.com in production

Env files:

.env

```
VITE_API_BASE_URL=http://localhost:3000
```

.env file contains default values.
We **must** prefix with **VITE_** otherwise the vars won't be loaded.

.env.production

```
VITE_API_BASE_URL=https://www.foo.com
```

.env.production overrides .env in production environment
(Vite uses the dotenv package which takes care of this for us)

In your frontend code:

```
const API_BASE_URL = import.meta.env.VITE_API_BASE_URL;
```

Read env variables from here in Vite apps

```
const response = await fetch(`${API_BASE_URL}/api/helloworld`);
```

Will be http://localhost:3000/api/helloworld in dev, and https://www.foo.com/api/helloworld in prod.

Use of env files

- Example:
  - Your dev environment backend is http://localhost:3000
  - You'll be **serving your frontend from your backend** in production (see next slide)

Env files:

.env

```
VITE_API_BASE_URL=http://localhost:3000
```

.env file contains default values.

.env.production

```
VITE_API_BASE_URL=
```

.env.production overrides .env in production environment. We're setting a blank string this time…

In your frontend code:

```
const API_BASE_URL = import.meta.env.VITE_API_BASE_URL;

const response = await fetch(`${API_BASE_URL}/api/helloworld`);
```

Will be http://localhost:3000/api/helloworld in dev, and just /api/helloworld in prod.

In a production environment…

- We can create a production-ready build of our Vite apps by using the `npm build` script.
  - This creates a folder called `dist`, which contains our production code
- If we like, we can serve the contents of this folder directly from our backend.

In a production environment…

- Adding this code to our Express backend will cause it to:
  - Serve all files in the dist directory statically
  - Offer up "index.html" when it receives an unknown GET request (i.e. makes routing work properly)
- … But only in production!

```
if (process.env.NODE_ENV === 'production') {
    console.log('Running in production!');

    app.use(express.static(path.join(__dirname, '../../frontend/dist')));

    app.get('*', (req, res) => {
        res.sendFile(path.join(__dirname, '../../frontend/dist/index.html'));
    });
}
```

Note: *Paths here assume the folder structure on the previous slide.*

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Three

Consuming REST APIs from React

Consuming REST APIs

- Many different possibilities for how we do this, depending on what our requirements are
  - What do we need to do? Full set of CRUD operations? Or a smaller subset? Or something else?
  - Is it likely that the data we GET from the server changes often? If it does, is it important that we show the changes immediately? Or can it wait for a browser refresh?
  - Do we need to show some UI elements (e.g. “loading” bar) while data is being loaded? If communication is successful? Unsuccessful?
  - How often do we need to update data on the server?
  - How likely is it that the app will be used in a poor-network environment? What “level of service” of the app is sufficient in these cases?
    - What if the app is entirely offline at some point? Do we want / need to handle this?

GET requests

- Simple case:
  - We're sending a GET request
  - One request per page load
  - Not concerned with error handling
  - Users must refresh the page to update
- We will use `useEffect()` to trigger a fetch / axios call.

GET requests with useEffect()


```
const [articles, setArticles] = useState([]);
```

A place to store the data

```
useEffect(() => {
    axios.get('${API_BASE_URL}/api/articles')
        .then(response => setArticles(response.data));
});
```

A side-effect that will issue a GET request to the given URL. When the response is received, updates the state (`setArticles()` in this case), causing the component to re-render and show the newly downloaded data.

What's the problem with this solution?

GET requests with useEfffect()


```
const [articles, setArticles] = useState([]);
```

A place to store the data

```
useEffect(() => {

    axios.get('${API_BASE_URL}/api/articles')
        .then(response => setArticles(response.data));

}, [] );
```

A side-effect that will issue a GET request to the given URL. When the response is received, updates the state (`setArticles()` in this case), causing the component to re-render and show the newly downloaded data.

- **Recall:** The optional second argument to `useEffect()` is an array. If supplied, then the side-effect function will only be called after re-rendering if any of the elements in the array have changed.
- If we supply an *empty array*, then none of the elements in that array could change (obviously!), therefore the side-effect will never trigger more than once.

Generalizing the operation

- We can encapsulate the functionality on the previous slide in a custom hook, so we can reuse it where other data needs to be fetched.

```
export default function useGet(url, initialState) {

    const [data, setData] = useState(initialState);

    useEffect(() => {
        axios.get(url)
            .then(response => setData(response.data));
    }, [url]);

    return data;
}
```

Generalizing the operation

- The same hook, rewritten to use `async` / `await`

```
export default function useGet(url, initialState = null) {

    const [data, setData] = useState(initialState);

    useEffect(() => {

        async function fetchData() {
            const response = await axios.get(url);
            setData(response.data);
        }

        fetchData();
    }, [url]);

    return data;
}
```

By the specification of `useEffect()` the effect function can't be async ☹

But we ***can*** write an async function here…

…and then immediately call it (or other async functions)!

Only reload if the URL changes.

Extending the hook

- We can extend our custom hook on the previous slide in any number of ways
  - E.g. a “loading” status
  - Error handling
  - Updating the URL
  - Handling query params
  - Manual refreshes (e.g. when clicking a button)
  - Automatic refreshes (timers?)
  - Custom code execution when complete
  - Etc…

Extending the hook with “loading” functionality


```
export default function useGet(url, initialState = null) {

    const [data, setData] = useState(initialState);
    const [isLoading, setLoading] = useState(false);

    useEffect(() => {
        async function fetchData() {
            setLoading(true);
            const response = await axios.get(url);
            setData(response.data);
            setLoading(false);
        }
        fetchData();
    }, []);

    return { data, isLoading };
}
```

Resources

- Express.js
- fetch()
- Axios
- Environment variables / .env files
  - In Vite (frontend)
  - In Nodejs/Express (backend)