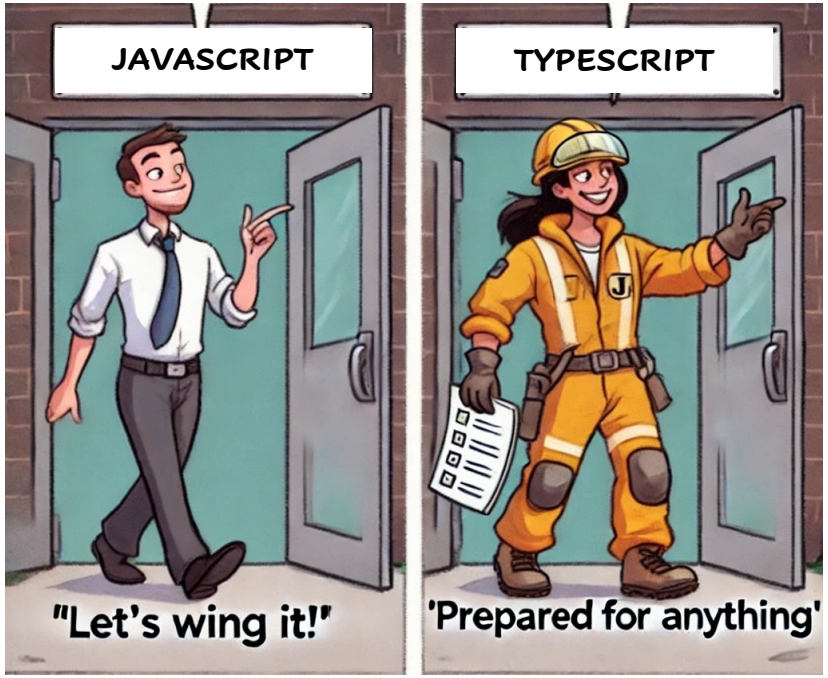


TypeScript & Advanced CSS

CSC309

Kianoosh Abbasi

UNIVERSITY OF TORONTO

Fall 2024

React so far

- Components, state, props
- Integration with Next.js
  - Monolithic project
- Hooks and API calls

This session

- Navigation with Next.js
- Global state and context
- Type safety with TypeScript
- Advanced CSS
  - Tailwind classes

Navigation

- You might need a URL change via code
- Example: If response is 401, redirect to the login page
- Like `window.location.replace()` in regular JS
- Via Next router:

```
let router = useRouter();
router.push("/login")
```

Arguments

- Parameters can both be defined as URL args (part of the path) or query params (key-value pairs added after ? in URL)

- URL args defined in the file name
  - e.g., [storeId].jsx

- Can be accessed via `router.query`

```
const { storeId } = router.query;
```

Links

- Like the familiar <a> tag, but without a browser reload

- Usage

```
<Link href="/watch"> watch </Link>
```

- Important: Import Link and Router from Next, not React

```
import Link from 'next/link'
import { useRouter } from 'next/router'
```

Prop drilling

- Passing state down to children can be quite cumbersome
- Example: The component that fires the request is a deep child button
  - You need to pass both the state and its setter function all the way down

Global state

- A global state is can be a great alternative

- Accessible everywhere!
  - No need to pass things all the way down

- Like global variables, don't use them for everything!
  - Makes your code dirty and harder to understand
  - Makes component re-use harder

Context

- React's way to handle global state
- Create state variables and put them and/or setters in a context
- Everything inside the context is accessible within its provider

Context

- Create the context (usually in a separate file)

```
export const TestContext = createContext({
    var1: null, var2: null,
});
```

- Put a default initial value for every variable that you will include in your context

Provider

- Create an object
```
const myObject = { var1: 1, var2: 2 };
```

- Put a provider around the parent component and pass the object
```
<TestContext.Provider value={myObject}>
    …
</TestContext.Provider>
```

- At any descendent, you can access the context object
```
const { var1, var2 } = useContext(TestContext)
```

- More information:
  https://dmitripavlutin.com/react-context-and-usecontext/

Why context is so great?

- Great way to store data that is used by many components, or it set and read in very different components
  - e.g, account info, profile data, etc.

- Create a context for each set of relevant variables and their setters

Context example

```
export function useAPIContext() {

    const [deployment, setDeployment] = useState([]);

    const [servers, setServers] = useState([]);

    const [applications, setApplications] = useState([]);

    const [applicationStatus, setApplicationStatus] = useState([]);

    const [availableLogDates, setAvailableLogDates] = useState([]);

    return {
        deployment,
        setDeployment,
        servers,
        setServers,
        applications,
        setApplications,
        applicationStatus,
        setApplicationStatus,
        availableLogDates,
        setAvailableLogDates,
    };
```

```
ReactDOM.render(
  <React.StrictMode>
      <APIContext.Provider value={useAPIContext()}>
          <ControlPanel/>
      </APIContext.Provider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

Codes by Myles Thiessen. https://thiessem.ca

Type safety

Type system

- Static vs dynamic typing
  - Static Typing: Types are checked at compile-time (e.g., C, Java).
  - Dynamic Typing: Types are checked at runtime – you can change a variable's type (e.g., Python).

- Strong vs weak typing
  - Strong Typing: Enforces strict rules about how types are used and combined (e.g., Java, Python)
  - Weak Typing: Flexible about type conversions, often leading to implicit type coercion.

- JavaScript types (recap):
  - number, string, boolean, object, function, undefined

- JavaScript is both dynamically and weakly typed
  - Can re-assign variables to different types
  - Automatically converts types in unexpected ways to avoid crashing

    `1 + "2"` results in `"12"`

    `5 == "5"` results in `true`

    `"0" == false` results in `true`

    `[] + []` results in `""`

    `[] + {}` results in `[object Object]`

Implications

- Plain JavaScript code is very error prone
- Having no typing makes the code unreadable for other developers
  - And also for yourself (within 3-4 weeks)
- It quickly becomes a mess and attracts bugs!

TypeScript

- Invented by Microsoft in 2012
- Superset of JavaScript:
  - Adds typing to the language
- Has no runtime effect!
  - Compiles to JavaScript

Add TypeScript to Next.js projects

- Simply create an empty file named tsconfig.json and restart the server.
- Next.js will automatically fill it up. Copy existing configs from jsconfig.json.
- Rename a file from js/jsx to ts/tsx and enjoy!
- Note: You can add TypeScript to all Node projects
  - Not limited to Next.js projects.
  - Visit https://dev.to/bhaeussermann/adding-typescript-support-to-your-nodejs-project-3bfm

TypeScript

Statically typed

```
let name = 'Alice'
name = 42 // Error: Type 'number' is not assignable to type 'string'
```

Strongly typed

```
const num = 10
const str = '20'
const result = num + str // Error: Operator '+' cannot be applied to types 'number' and 'string'
```

TypeScript benefits

- Improved code quality
- Catch errors during development, not at runtime
- Better collaboration in large teams with clear types
- Better tooling and autocompletion in IDEs

TypeScript syntax

- Type declaration

```
let message: string = "Hello, TypeScript!"
function greet(name: string): string {
  return `Hello, ${name}`
}
```

- Type inference also works

```
let count = 42; // infers type 'number'
```

Type system

- Primitive Types:
  - JavaScript primitive types (string, number, boolean, etc.)
  - Plus additional types: any, unknown, void, never.
  - Array and tuple: `number[]`, `[string, number]`
  - Enums
  - Optionals: `function greet(name?: string)…`

- Type aliases:

```
type ID = string | number
```

Type system

- Interfaces and generic types

```
interface ModalProps {
  text: string
  image?: string
  autoHide: boolean
}

const Modal: React.FC<ModalProps> = (props) => {
  const [loading, setLoading] = useState<boolean>(false)
}
```

TypeScript notes

- TypeScript works alongside JavaScript
  - All files do not have to be converted TypeScript

- Suppression
  - The `any` type is a wildcard that suppresses type checking
  - Use `@ts-ignore` to disable TypeScript for a line
  - Discouraged. Use only if it's absolutely necessary.

TypeScript notes

- Remember: all these checks are compile-time. None of them have any impacts at runtime!
- Runtime code is again plain JavaScript
- To check types at runtime, use `typeof` and `instanceof`
  - Only work for JavaScript types and classes, respectively

Advanced CSS

Traditional CSS

- CSS bloat
  - CSS files grow very big with a lot of unused styles
- Specificity war
  - Overly complex rules for CSS precedence
- CSS frameworks (e.g., bootstrap, material, etc.)
  - Leads to websites that look similar
- Context switching between JS and CSS files

Tailwind CSS

Visit https://tailwindcss.com/docs

- Replaces CSS styles with utility classes

- Example

```
<button className="bg-blue-500 text-white font-bold py-2
px-4 rounded">
  Click Me
</button>
```

- Each class adds the corresponding CSS styles to the element

Installation

Visit https://tailwindcss.com/docs/guides/nextjs

- Install via `npm install tailwindcss postcss autoprefixer`
- Run `npx tailwindcss init -p` to generate config files
  - Generates tailwind.config.js and postcss.config.js
- Add all JS, JSX, TS, and TSX file globs to `content` in tailwind.config.js
  - This will tell Tailwind to look for utility classes in those files
- Add the following lines to globals.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Power of tailwind

- Arbitrary values

```
"w-[50%] text-[#ff6347]"
```

- Dark and light modes

```
"bg-white dark:bg-gray-800"
```

- Responsive styles

```
"p-4 sm:p-6 md:p-8"
```

- CSS interoperability

```
h1 {
  @apply text-2xl font-bold;
}
```

- Custom themes

  Define theme colors, font, sizes in tailwind.config.js

  See https://tailwindcss.com/docs/theme

Responsive design

- Should render well in different devices
  - Wide screeners, laptops, tablets, smart phones
- Tailwind makes having responsive styles easy
- General tip: avoid absolute lengths
  - The most responsive unit is rem
  - Good news: tailwind units translate into rem!
    - e.g., `pt-4` becomes `padding-top: 1rem;`
- Flex and grid layouts can be helpful!

Flex

- Horizontally or vertically places items inside the parent element (aka container)
- add `class="flex"` (or `"flex flex-col"` for vertical) to container
- To wrap items on overflow, add `"flex-wrap"`
- Handle spacing between items:
  `justify-center`, `justify-between`, `justify-around`, `justify-evenly`, etc.

Flex items

Visit https://tailwindcss.com/docs/flex

- Control how much space each item takes if there's extra space (or too little space)

- Use `flex-1` to take whatever space left
  - Use case: Navbar with some links on one end, some on the other

- Divide it between multiple elements with the same class

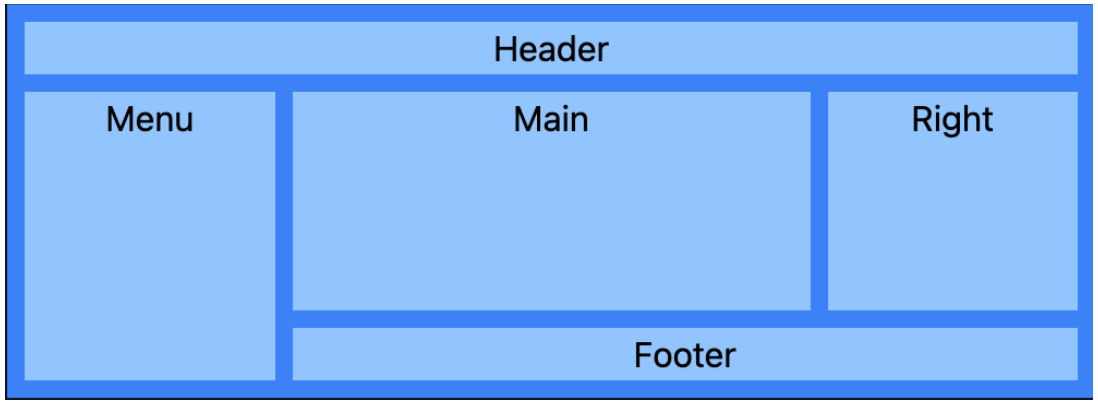
Grid layout

- Specify in the container
  `class="grid grid-cols-3 gap-4"`

- Specify how much space each item needs
  `class="col-span-2"`

- Often need responsive grids
  You will need fewer columns in small devices
  `class="grid sm:grid-cols-2 md:grid-cols-3 gap-4"`

Grid example




```
<div className="grid bg-blue-500 grid-cols-4 gap-2 p-2 text-black text-center">
  <div className="bg-blue-300 col-span-4">Header</div>
  <div className="bg-blue-300 row-span-2">Menu</div>
  <div className="bg-blue-300 col-span-2 h-[100px]">Main</div>
  <div className="bg-blue-300">Right</div>
  <div className="bg-blue-300 col-span-3">Footer</div>
</div>
```

Tailwind notes

- While it's a great tool, you will have to be cautious!
- className bloat!
  - Long, often repetitive classes
- Re-used classes is often a signal for extracting new components
- Use `@apply` to move the classes to CSS

Next session

- Concept of isolation
- Intro to Docker
  - DockerFile
  - Containers, images, registry
- Docker compose
- Course conclusion