Week 04 - Component Libraries, React Hooks and Data fetching

Filip Kaštovský

Outline

- Motivation (why we need component libraries)
- Creating a component library
- React hooks & data fetching

Articles

All Articles | Saved Articles

CO2 emissions: Train versus plane

When it comes to carbon dioxide (CO2) emissions, many people wonder if taking a train or a plane is better for the environment. The answer to this...

Read more

The Benefits of Composting and How to Get Started

Composting is the process of decomposing organic materials into a nutrient-rich soil-like substance that can be used as a fertilizer for...

Read more

CO2 emissions: Train versus plane

When it comes to carbon dioxide (CO2) emissions, many people wonder if taking a train or a plane is better for the environment. The answer to this...

Read more

The Benefits of Composting and How to Get Started

Composting is the process of decomposing organic materials into a nutrient-rich soil-like substance that can be used as a fertilizer for...

Read more →

CO2 emissions: Train versus plane

When it comes to carbon dioxide (CO2) emissions, many people wonder if taking a train or a plane is better for the environment. The answer to this...

Read more

The Benefits of Composting and How to Get Started

Composting is the process of decomposing organic materials into a nutrient-rich soil-like substance that can be used as a fertilizer for...

Read more

A single file will only get you so far

```
const App: FC = () => {
  const { articles } = useArticles();

  return (
    <div className="app-layout">
      <header className="app-layout__header header">
        <h1 className="header__title">Articles</h1>
        <div className="tabs">
          <a className="tabs__tab">All articles</a>
          <a className="tabs__tab tabs__tab--active">Saved articles</a>
        </div>
      </header>
      <main className="app-layout__main">
        <section className="articles">
          {articles.map((article) => (
            <div className="card articles__article">
              <img className="card__image" src={article.image} />
              <h2 className="card__title">{article.title}</h2>
              <p className="card__text">{article.description}</p>
              <button className="card__action btn">Read mode</button>
            </div>
          ))}
        </section>
      </main>
```

Yes, it didn't fit on the slide

Yes, it didn't fit on the slide

```
      <nav className="app-layout__navbar navbar">
        <a className="navbar__item navbar__item--active">
          <BookIcon />
        </a>
        <a className="navbar__item">
          <TrophyIcon />
        </a>
        <a className="navbar__item">
          <GlobeIcon />
        </a>
        <a className="navbar__item">
          <UsersIcon />
        </a>
        <a className="navbar__item">
          <GearIcon />
        </a>
      </nav>
    </div>
  );
};

export default App;
```

Identifying components

Articles

All Articles | Saved Articles

CO2 emissions: Train versus plane

When it comes to carbon dioxide (CO2) emissions, many people wonder if taking a train or a plane is better for the environment. The answer to this...

Read more

The Benefits of Composting and How to Get Started

Composting is the process of decomposing organic materials into a nutrient-rich soil-like substance that can be used as a fertilizer for...

Read more

CO2 emissions: Train versus plane

When it comes to carbon dioxide (CO2) emissions, many people wonder if taking a train or a plane is better for the environment. The answer to this...

Read more

The Benefits of Composting and How to Get Started

Composting is the process of decomposing organic materials into a nutrient-rich soil-like substance that can be used as a fertilizer for...

Read more →

CO2 emissions: Train versus plane

When it comes to carbon dioxide (CO2) emissions, many people wonder if taking a train or a plane is better for the environment. The answer to this...

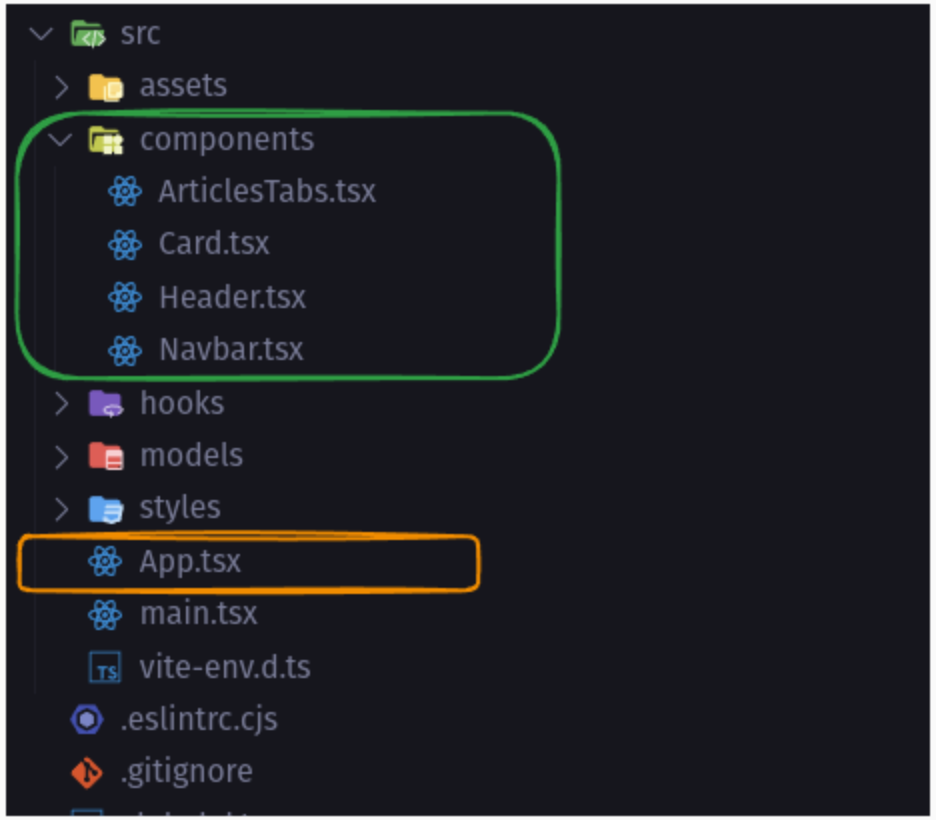
Read more

The Benefits of Composting and How to Get Started

Composting is the process of decomposing organic materials into a nutrient-rich soil-like substance that can be used as a fertilizer for...

Read more

Splitting code into components



Splitting code into components

```
const App: FC = () => {
  const { articles } = useArticles();

  return (
    <div className="app-layout">
      <Header className="app-layout__header" />
      <main className="app-layout__main">
        <section className="articles">
          {articles.map((article) => (
            <Card className="articles__article" {...article} />
          ))}
        </section>
      </main>
      <Navbar className="app-layout__navbar" />
    </div>
  );
};

export default App;
```

But thats only a single page, real websites have multiple pages!

Do I need to create a new `App.tsx` -like file for every page?

No!

But let's discuss a JSX superpower first! (that you've been using all along)

Recap: Passing props

- In React, parent components can **pass data** or **callback functions** to child components using `props`
- `props` act as arguments to a function (they technically are just boxed arguments)

```
type CardProps = {
  title: string;
  description: string;
};
```

```
const Card: FC<CardProps> = (props) => {
  return (
    <div className="card">
      <h2>{props.title}</h2>
      <p>{props.description}</p>
    </div>
  );
};
```

```
<Card title="Hello" description="World" />
```

React Slots

- `props` do not have to be just data! They can also be **React elements** (instances of components)
- We use a special type for annotating this kind of `props` - `ReactNode`
  - `ReactNode` is a type of everything **renderable** in React
- This pattern of injecting markup is called **slots** in many different frameworks, including React

```
type CardProps = {
  title: ReactNode;
  description: ReactNode;
};
```

React Slots

- aaaand allows to generalize our `Card` component like so:

```
const Card: FC<CardProps> = (props) => {
  return (
    <div className="card">
      {props.title}
      {props.description}
    </div>
  );
};
```

```
<Card title={<h2>Hello</h2>} description={<p>World</p>} />
```

Cool, right?

Recap: React Slots

Instead of passing data to be rendered in the markup, we can pass the markup itself!

- Utilizes `ReactNode` type
- **slots** allow high level of reusability for components, where it would otherwise be impossible
- Correctly using **slots** is **key** for maintaining good performance in React (more on that some other time!)

Pop quiz

```
<Card title="Hello" description={null} />
```

> Given our *slotted* `Card` component, is this still valid?

```
const Card: FC<CardProps> = (props) => {
  return (
    <div className="card">
      {props.title}
      {props.description}
    </div>
  );
};
```

The special prop you've been using all along - `children`

JSX children

- components can have a special prop called `children` defined with either
  - wrapping your props with `PropsWithChildren<YourProps>`
  - by adding `children: ReactNode` to the props type
- `children` is a special prop that contains the **content** between the opening and closing tags of the component
- allows for easy nesting of elements

JSX children

```
type CardProps = {
  title: string;
  description: string;
  children: ReactNode;
};
```

```
const Card: FC<CardProps> = (props) => {
  return (
    <div className="card">
      <h2>{props.title}</h2>
      <p>{props.description}</p>
      {props.children}
    </div>
  );
};
```

JSX children - `PropsWithChildren<T>`

```
type CardProps = {
  title: string;
  description: string;
};
```

```
const Card: FC<PropsWithChildren<CardProps>> = (props) => {
  return (
    <div className="card">
      <h2>{props.title}</h2>
      <p>{props.description}</p>
      {props.children}
    </div>
  );
};
```

JSX Children - usage

```
<Card title="Hello" description="World">
  <button>Click me</button>
  <img src="..." alt="some image" />
  <div>Some other content</div>
</Card>
```

> Looks familiar? How about using a native tag:

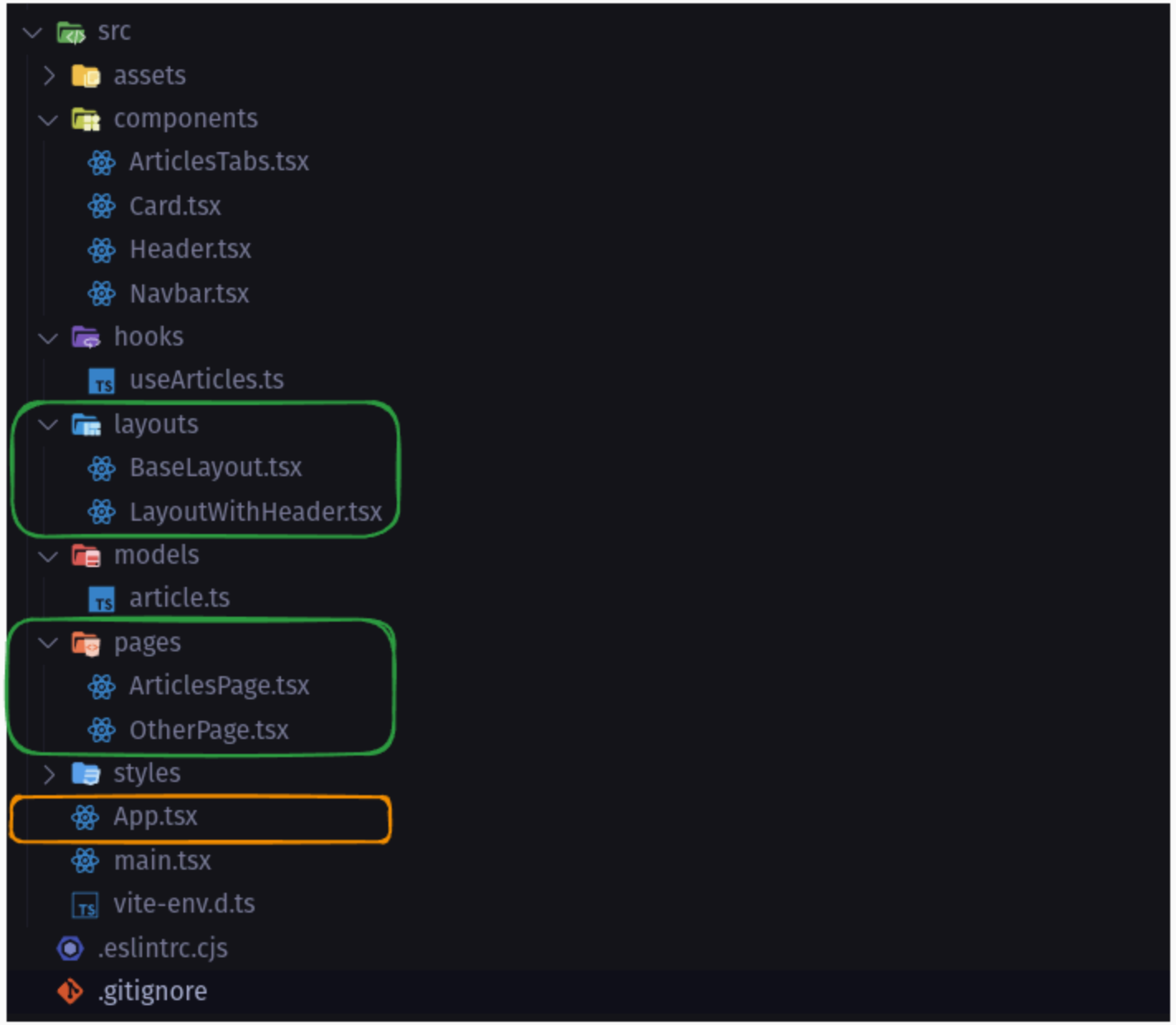
```
<section>
  <button>Click me</button>
  <img src="..." alt="some image" />
  <div>Some other content</div>
<section>
```

- Your components can now render other elements!

Layouts

Using slots and children, we can create a layout component that can render any content...

```
const BaseLayout: FC<BaseLayoutProps> = ({ header, children }) => {
  return (
    <div className="app-layout">
      {header}
      <main className="app-layout__main">{children}</main>
      <Navbar className="app-layout__navbar" />
    </div>
  );
};
```


src
assets
components
ArticlesTabs.tsx
Card.tsx
Header.tsx
Navbar.tsx
hooks
useArticles.ts
layouts
BaseLayout.tsx
LayoutWithHeader.tsx
models
article.ts
pages
ArticlesPage.tsx
OtherPage.tsx
styles
App.tsx
main.tsx
vite-env.d.ts
.eslintrc.cjs
.gitignore

Pages

- Pages are top level components, usually rended by a `router` (more on that later)
- Page structure (generally):
  - Layout
  - Page content

Articles

All Articles | Saved Articles

CO2 emissions: Train versus plane

When it comes to carbon dioxide (CO2) emissions, many people wonder if taking a train or a plane is better for the environment. The answer to this...

Read more

The Benefits of Composting and How to Get Started

Composting is the process of decomposing organic materials into a nutrient-rich soil-like substance that can be used as a fertilizer for...

Read more

CO2 emissions: Train versus plane

When it comes to carbon dioxide (CO2) emissions, many people wonder if taking a train or a plane is better for the environment. The answer to this...

Read more

The Benefits of Composting and How to Get Started

Composting is the process of decomposing organic materials into a nutrient-rich soil-like substance that can be used as a fertilizer for...

Read more →

CO2 emissions: Train versus plane

When it comes to carbon dioxide (CO2) emissions, many people wonder if taking a train or a plane is better for the environment. The answer to this...

Read more

The Benefits of Composting and How to Get Started

Composting is the process of decomposing organic materials into a nutrient-rich soil-like substance that can be used as a fertilizer for...

Read more

Pages

```
const ArticlesPage: FC<ArticlesProps> = (props) => {
  const { className, ...otherProps } = props;

  const { articles } = useArticles();

  return (
    <BaseLayout header={<Header className="app-layout__header" />}>
      <section {...otherProps} className={clsx("articles", className)}>
        {articles.map((article) => (
          <Card className="articles__article" {...article} />
        ))}
      </section>
    </BaseLayout>
  );
};
```

PRO TIP #1: create a separate component from the `section` tag for even cleaner code!
PRO TIP #2: extract commonly used layouts into components and use those! (`LayoutWithHeader` or `LayoutWithXYSidebar`)

What about multiple projects?

A company generally sticks to a single design language, but has multiple projects...

What about multiple projects?

- How do we
  - make sure our UI elements are consistent across applications?
  - avoid reinventing the wheel every time we start a new project?
  - make sure our components are tested and documented?

A Component Library!

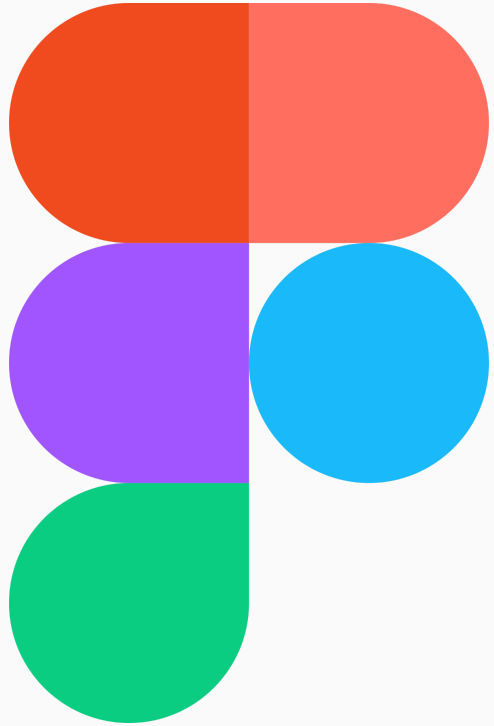
- A collection of reusable components, separated from the application code
- Can be used across multiple projects
- Usually published as a **package** to a package manager (npm, yarn)
    - or used as a **package** in a *monorepo* (foreshadowing...)

Component Libraries - a good library needs:

- To be part of a **Design system** - a set of rules and guidelines for the design and development of UI
- **Documentation** - how to use the components, what props they accept, what they look like
- **Open closed principle** - components should be open for extension, but closed for modification
  - UI components should be easily customizable
- **Testing** - components should be tested to ensure they work as expected
- **Accessibility** - components should be accessible to all users
- **Performance** - components should be performant

Component Libraries - Design system

- A UI/UX designer creates a design system
- Style guide
  - Color palette
  - Typography
  - Spacing
- **Component library**
  - Buttons
  - Inputs
  - Cards
  - Modals
  - etc.
- **Pattern library**
  - Rules on how to combine components to reusable patterns



Component Libraries - things to watch out for

- Introduces dependencies
  - Changing a component has a ripple effect on all projects
    - Remedied by correct **versioning**
- Up front time investment
  - Creating a component library takes time
  - But saves time in the long run

Pop quiz

Given this entry in a package file:

```
"my-component-library": "^1.1.5"
```

Is it possible for the following versions to be installed?

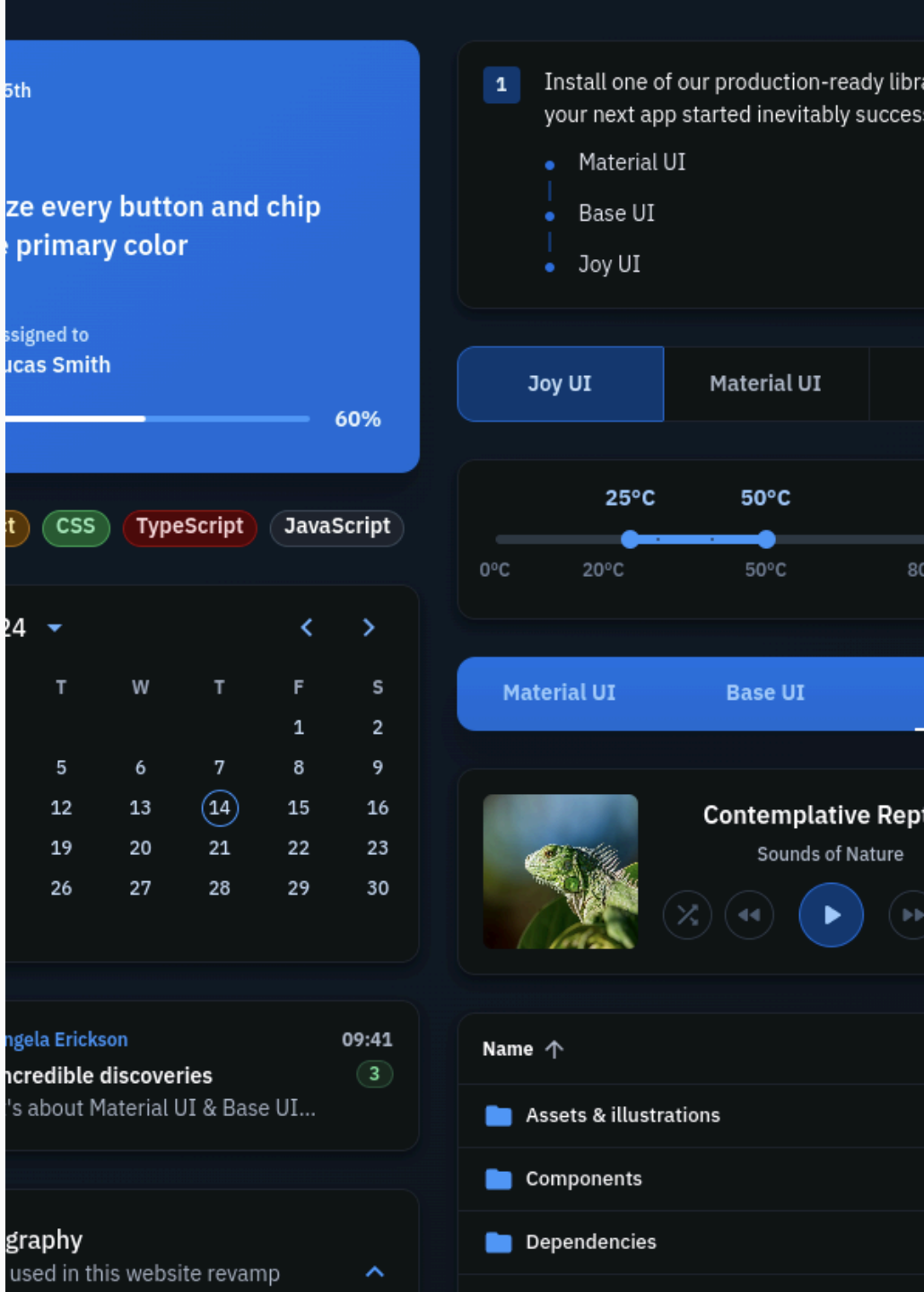
> a) 1.1.5
> b) 2.1.5
> c) 1.1.10
> d) 1.1.4
> e) 1.2.1

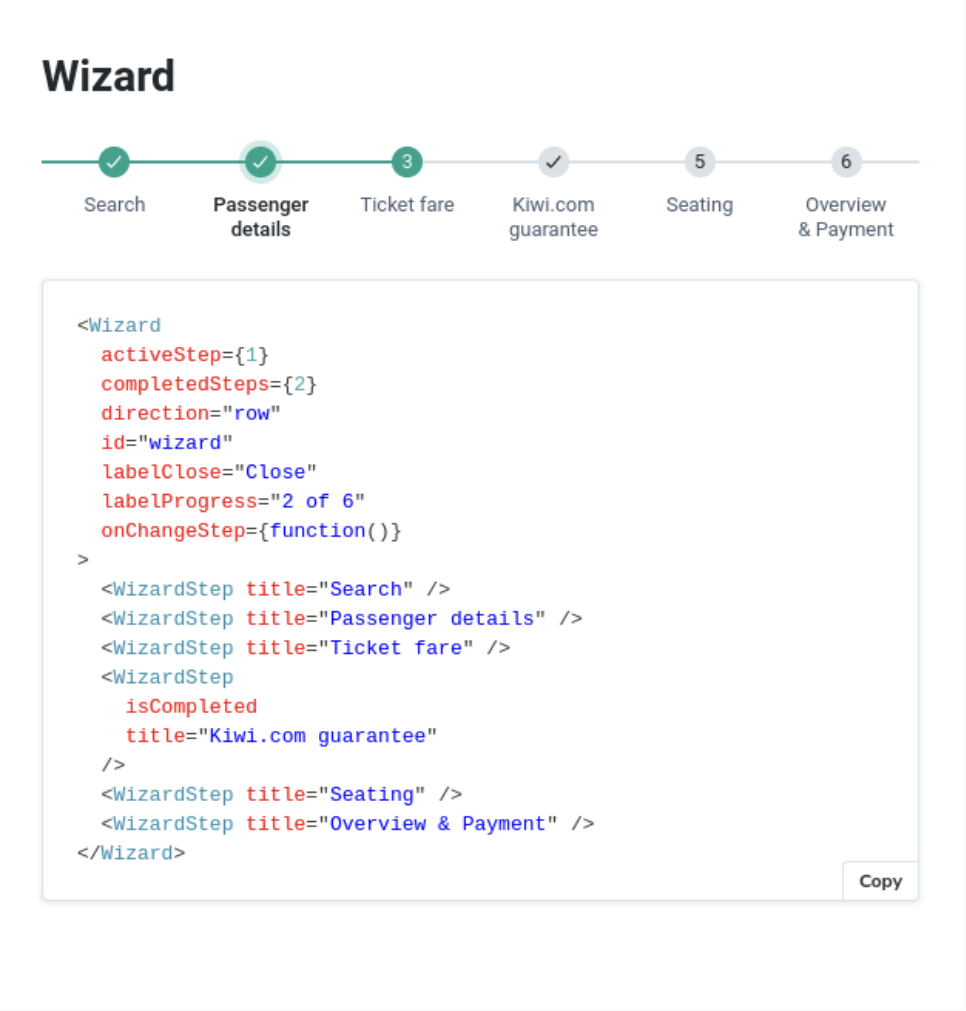
Off-the-shelf component libraries

- Material UI, Ant Design, Tailwind UI
- **Pros**
  - Well tested
  - Well documented
  - Widely used
  - **No time investment needed**
- **Cons**:
  - Your project look *generic*
  - Library learning curve
  - *Fighting the framework* - you have to adapt your design to the library

> "Material UI - pro ty kdo chtějí ať každá stránka vypadá stejně na..."
>
> - Tomáš Sedláček

TLDR: Use for internal tools, admin panels, etc.





How to create a component library?

Storybook

- Industry standard tool for developing UI components in isolation
- Supports many frameworks (React, Vue, Angular, Svelte, etc.)
- Easily integrates with exiting projects
  - just add *.story.tsx* files
- Very extensible with plugins
- Version 7 supports `vite` (used to be a pain to setup)
- Storybooks can be of various quality
  - Kiwi Orbit
  - MS Fluent UI

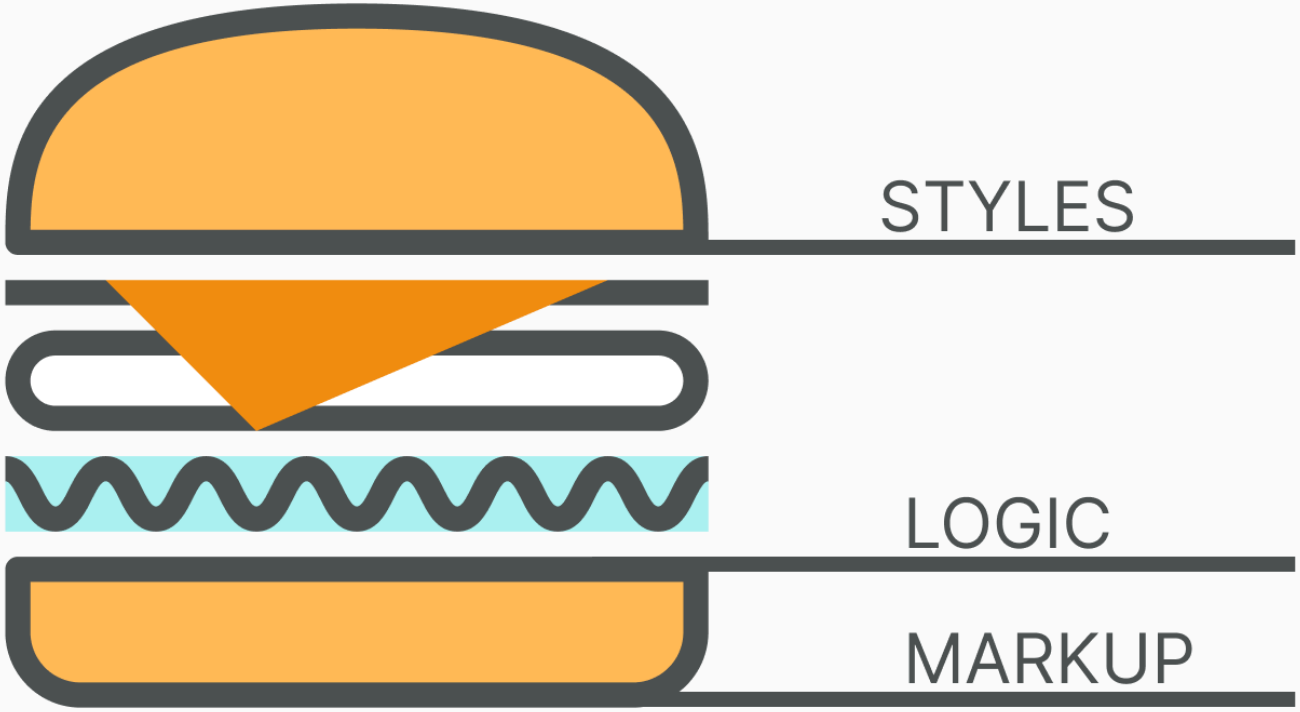
```
import { number, array, select, text } from "@storybook/addon-knobs";
import { action } from "@storybook/addon-actions";

export const Default = () => {
  const direction = select("direction", ["row", "column"], "row");
  const labelClose = text("labelClose", "Close");
  const labelProgress = text("labelProgress", "of");

  return (
    <Wizard
      id="wizard"
      direction={direction}
      completedSteps={2}
      activeStep={1}
      labelClose={labelClose}
      labelProgress={`2 ${labelProgress} 6`}
      onChangeStep={action("onChangeStep")}
    >
      <WizardStep title="Search" />
      <WizardStep title="Passenger details" />
      <WizardStep title="Ticket fare" />
      <WizardStep title="Kiwi.com guarantee" isCompleted />
      <WizardStep title="Seating" />
      <WizardStep title="Overview & Payment" />
    </Wizard>
  );
};
```

React component anatomy & ideology

- **Markup** - the component's markup
- **Logic** - the JavaScript that makes the component work
- **Styles** - the CSS that makes the component look good



```tsx
// .tsx
export const Switch = () => {
  const [checked, setChecked] = useState(false);

  useEffect(() => {
    console.log("Switch state changed. Checked:", checked);
  }, [checked]);

  return (
    <div className="switch">
      <input
        type="checkbox"
        checked={checked}
        onChange={(e) => setChecked(e.target.checked)}
      />
      <div className="switch__slider"></div>
    </div>
  );
};
```

```css
/* .css */
.switch {
  /* ... some css; */
}
...
```

Common patterns - Smart and dumb components

- **Smart components** (containers)
  - contain the logic
  - pass data to dumb components
  - handle events
  - fetch data
- **Dumb components** (presentational)
  - receive data from smart components
  - render the markup
  - emit events upstream

Emerging pattern - Combining markup and styles

- **Tailwind CSS** - styles are collocated with the markup, this allows faster iteration
- Sharing styles is now an issue! A **component library** is a must for scalable use of Tailwind CSS.

```
<figure className="bg-slate-100 rounded-xl p-8 dark:bg-slate-800">
  <img className="w-24 h-24 rounded-full mx-auto" src="/sarah-dayan.jpg" alt="" width="384" height="512">
  <div className="pt-6 text-center space-y-4">
    <blockquote>
      <p className="text-lg font-medium">
        “Tailwind CSS is the only framework that I've seen scale
        on large teams. It’s easy to customize, adapts to any design,
        and the build size is tiny.”
      </p>
    </blockquote>
    <figcaption className="font-medium">
      <div className="text-sky-500 dark:text-sky-400">
        Sarah Dayan
      </div>
      <div className="text-slate-700 dark:text-slate-500">
        Staff Engineer, Algolia
      </div>
    </figcaption>
  </div>
</figure>
```

React Hooks and Data fetching

useState

- create local state in a functional component

~~useEffect~~ useFootGun

- run side effects and synchronize with the outside world

useMemo/useCallback

- memoize values and functions

useState

```
const [value, setValue] = useState<Type>(initialValue | () => initialValue);
```

- `value` - the current state
- `setValue` - function to update the state

```
export const Counter: FC = () => {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
};
```

useState - functional update

- Updates to the `value` are **asynchronous**, React choses when to apply them
- Running multiple `setValue` calls in a row can lead to unexpected results
- The next value is computed from the `value` constant, so this set of calls always produces the same result:

```
setValue(value + 1);
setValue(value + 1);
setValue(value + 1);
```

> `value` only gets updated once

For this reason, `setValue` also accepts a function as an argument, which aplies the update to the current state, which can also be changed **between** renders

```
setValue((prev) => prev + 1);
setValue((prev) => prev + 1);
setValue((prev) => prev + 1);
```

> Now works as expected

Prefer functional updates when updating state based on the previous state

useEffect

```
useEffect(() => {
  // run some code
  return () => {
    // cleanup after the effect
  };
}, [dependencies]);
```

- Has wiiieeerd behavior
- Shoots you in the foot
- Do not use!

Used for synchronization with world outside of React (libraries which only work with basic JavaScript, running code regardless of the stage of the render cycle, etc.)

React Hooks - dependency array

- When one of the values in the dependency array changes, the code in the hook is re-run
- If the dependency array is empty, the hook code will never rerun - code is executed once per component mount
- `Object.is` comparison

Always specifically enumerate the dependencies

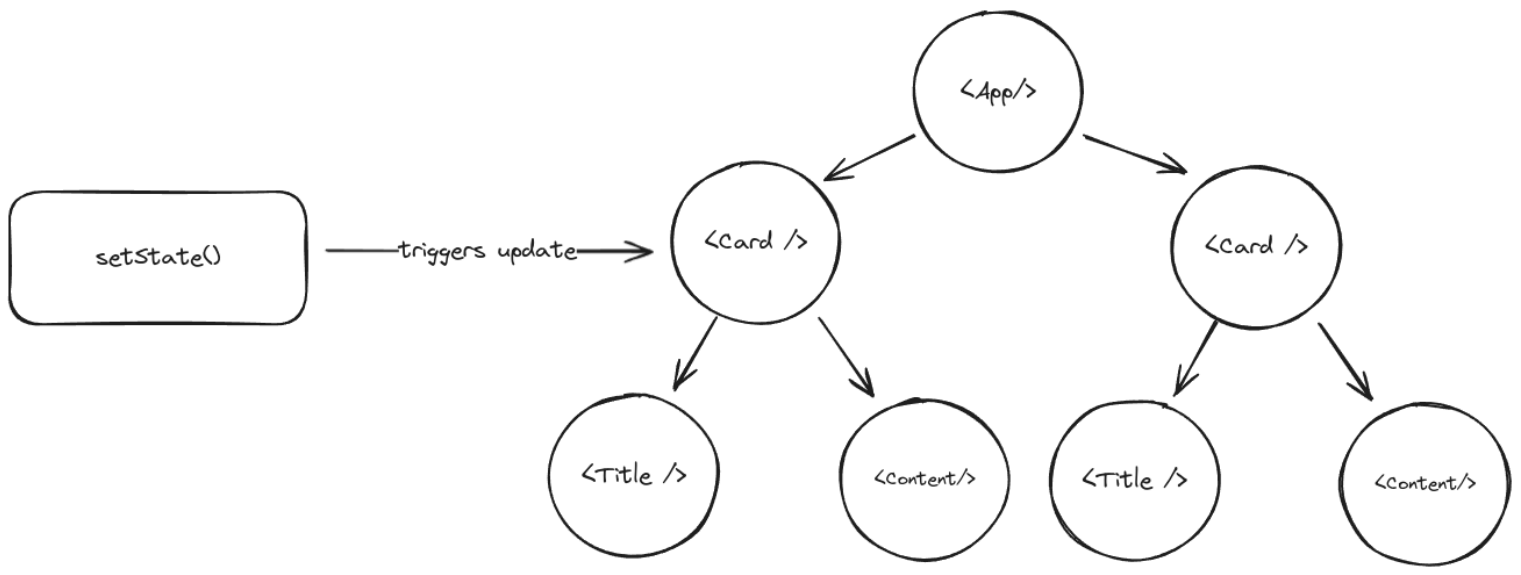
```
[foo, bar, baz];
```

These values will always cause updates:

```
[{}, [], new Set(), ...];
```

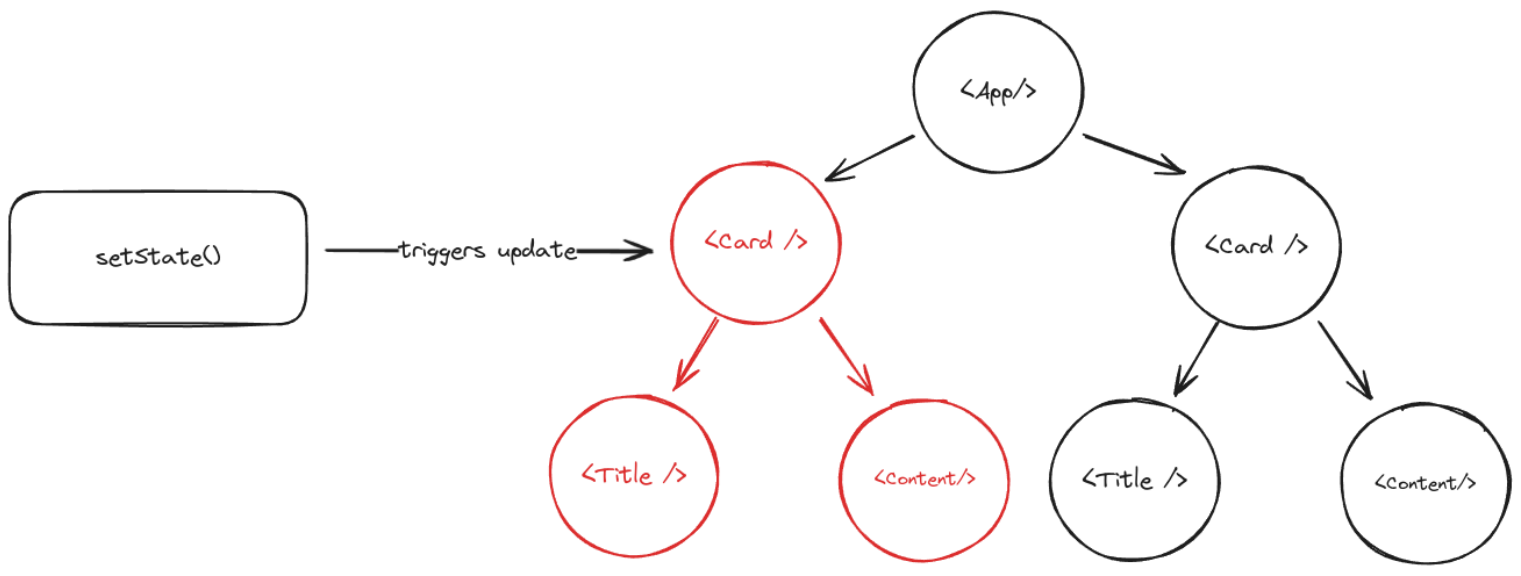
Component update model

- React maintains a *tree-like* structure for maintaining its VDOM
- Updates are handled by invalidating subtrees (*as rerenders in VDOM are cheap*)



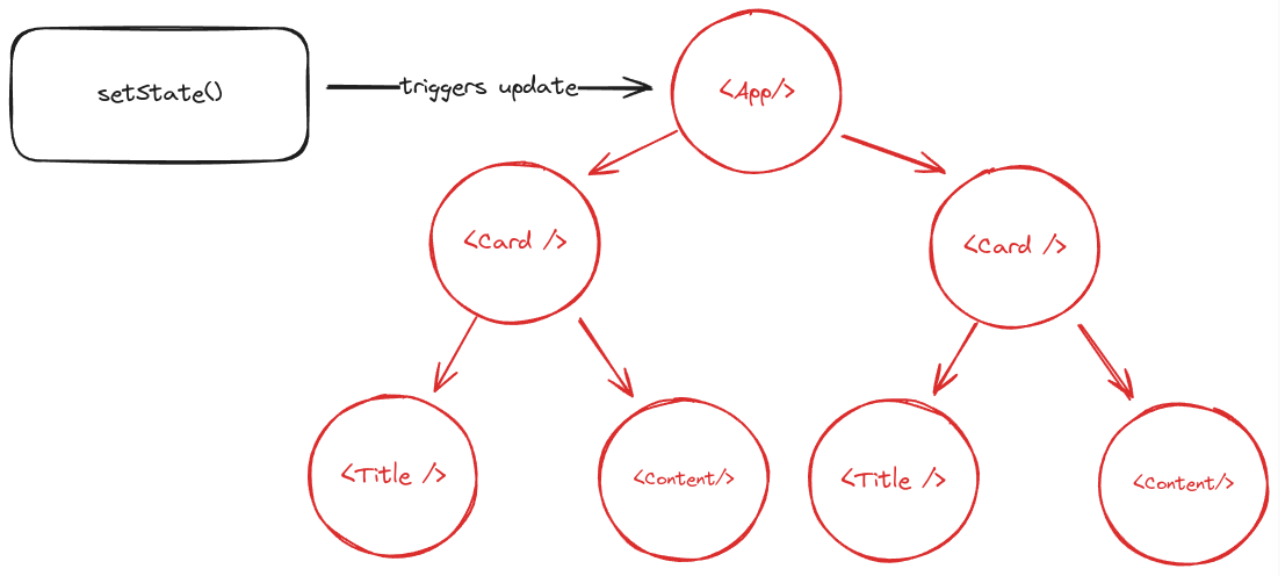
Component update model

- React maintains a *tree-like* structure for maintaining its VDOM
- Updates are handled by invalidating subtrees (*as rerenders in VDOM are cheap*)



Component update model

- React maintains a *tree-like* structure for maintaining its VDOM
- Updates are handled by invalidating subtrees (*as rerenders in VDOM are cheap*)



React Memo

- Every call to `useState` triggers an update of the subtree
- This behavior is opt-out via `React.memo`
- For large component trees, `React.memo` is a necessity

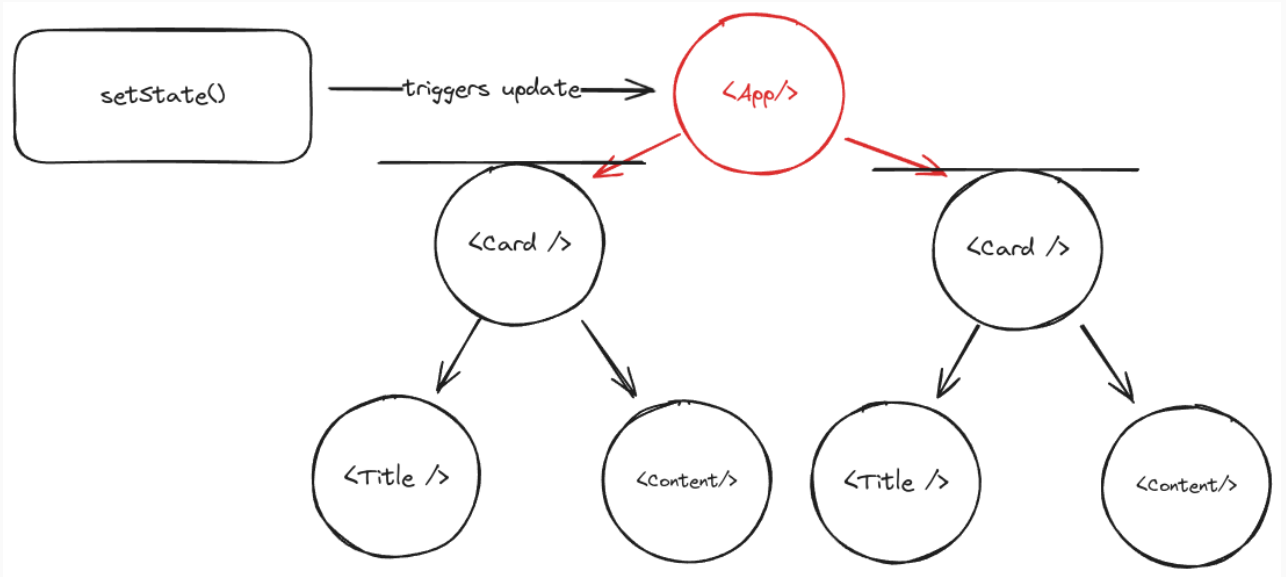
```
const MC = memo(MyComponent);
```

> Memo will compare current props with previous ones to see if they changed, and only then update the component and its subtree

```
<MC data={{}} />
```

```
<MC onClick={() => {}} />
```

> Memo is useless here! Why?



useMemo / useCallback

- Takes in a value, only recomputes it if its dependencies change
- Used as optimization for expensive computations (as React if not done properly **will rerun your components like crazy**)

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- `useMemo` however has one more use (and superpower) - **stabilising references across renders**

Stabilising references and derived state

Creating a non-primitive value will always cause React's equality check to fail, hindering performance and causing unexpected behaviors

> But I still want to pass objects as props! Or functions!

Common workaround (and a huge antipattern!):

```
const [firstName, setFirstName] = useState("John");
const [lastName, setLastName] = useState("Doe");
const [name, setName] = useState({ firstName, lastName });

useEffect(() => {
  setName({ firstName, lastName });
}, [firstName, lastName]);
```

Never duplicate state! Always keep a single source of truth.

This code also trigerrs re-rendering of the component twice, which is a performance hit in case of a larger component, or more complex stored state.

Stabilising references and derived state

For calculating referential values from others, use `useMemo`:

```
const [firstName, setFirstName] = useState("John");
const [lastName, setLastName] = useState("Doe");

const name = useMemo(() => ({ firstName, lastName }), [firstName, lastName]);
```

> `useMemo` will only recompute the value if `firstName` or `lastName` change, keeping the `name` reference pointing to the same object **unless its dependencies change**

Always list all the dependencies! Not one too many, not one too few. (eslint: `react-hooks/exhaustive-deps`)

useCallback

- Similar to `useMemo`, but for functions
- Uses `useMemo` under the hood

```
const handleClick = () => {
  console.log(`Button clicked ${count} times`);
};

<Component onClick={handleClick} />;
```

With `useCallback`

```
const handleClick = useCallback(() => {
  console.log(`Button clicked ${count} times`);
}, [count]);
// safe
<Component onClick={handleClick} />;
```

> PRO TIP: If your functions are not reliant on react state, they do not need to be memoized, **just define them outside of the component**!

Data fetching basics

- Most of the frontend applications today are data driven
  - fetching data from an API
  - displaying it to the user
  - allowing the user to interact with it, update it
- Modern browsers have a built-in API for fetching data over HTTP - `fetch`, that works without refreshing the page (AJAX)

```
fetch("https://api.example.com/data", options)
  .then((response) => response.json())
  .then((data) => console.log(data));
  .catch((error) => console.error(error));
```

> Fetch is very convenient, it returns a promise!

```
const getData = async () => {
  const response = await fetch("https://api.example.com/data", options);
  return response.json();
};
```

```
try {
  const data = await getData();
  console.log(data);
} catch (error) {
  console.error(error);
}
```

Axios

```
$ npm i axios
```

- `axios` is a popular library for making HTTP requests
  - `fetch` used to not be well supported in older browsers
- has a more convenient API
- has built-in support for interceptors, request and response transformations, etc.
- internally uses a lower level event API - `XMLHttpRequest`, allowing for more control (such as displaying file upload progress)

```
axios
  .get("https://api.example.com/data", options)
  .then((response) => console.log(response.data))
  .catch((error) => console.error(error));
```

```
const getData = () => axios.get("https://api.example.com/data", options);

try {
  const response = await getData();
  console.log(response.data);
} catch (error) {
  console.error(error);
}
```

Data fetching in React

Fetching data is a side effect, React is not very good at handling side effects (there is useEffect but...)

Ideally, we want to delegate this to something outside of React...

Tanstack Query

```
$ npm i @tanstack/react-query
$ npm i -D @tanstack/eslint-plugin-query
```

- `react-query` is a library for managing server state in your React applications
- its features include
  - caching
  - deduplicating requests
  - loading and error states
  - background updates
  - optimistic updates
  - pagination
  - and soo much more... (seriously, go read the docs)
- TLDR: It's a swiss army knife for data fetching, and it's very easy to use (over implementing all of this yourself)

Tanstack Query - usage

```
import { useQuery } from "@tanstack/react-query";

const queryClient = new QueryClient();

const App = () => {
  return (
    // Provide the client to your App
    <QueryClientProvider client={queryClient}>
      <Todos />
    </QueryClientProvider>
  );
};
```

Tanstack Query - Queries

- Used to `get` the data from the server
- Unless specified otherwise, the HTTP will run when the component mounts (skipToken)

```
const Todos = () => {
  const query = useQuery({ queryKey: ["data"], queryFn: getData });

  return (
    <div>
      <ul>
        {query.data?.map((todo) => (
          <li key={todo.id}>{todo.title}</li>
        ))}
      </ul>
    </div>
  );
};
```

- Each component referring with the same `queryKey` will share the same data
- If multiple components request the same data, the request will be deduplicated, only one request will be made

Tanstack Query - Mutations

- Updates the server state

```
const UpdateTodos = () => {
  // Access the client
  const queryClient = useQueryClient();

  // Mutations
  const mutation = useMutation({
    mutationFn: updateData,
    onSuccess: () => {
      // Invalidate and refetch
      queryClient.invalidateQueries({ queryKey: ["data"] });
    },
  });

  const handleClick = useCallback(() => {
    mutation.mutate({
      id: Date.now(),
      title: "Do Laundry",
    });
  }, [mutation]);

  return <button onClick={handleClick}>Add Todo</button>;
};
```

Resources

Figma logo
Material UI site screenshot

AntDesign
Tailwind UI
Kiwi Orbit
MS Fluent UI
React Query
React Query - skipToken
React docs
MDN - Fetch
Axios