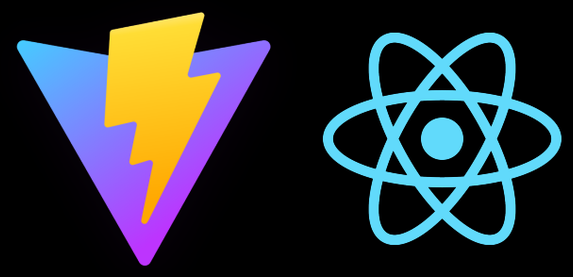


Typescript

Adds optional static typing to Javascript

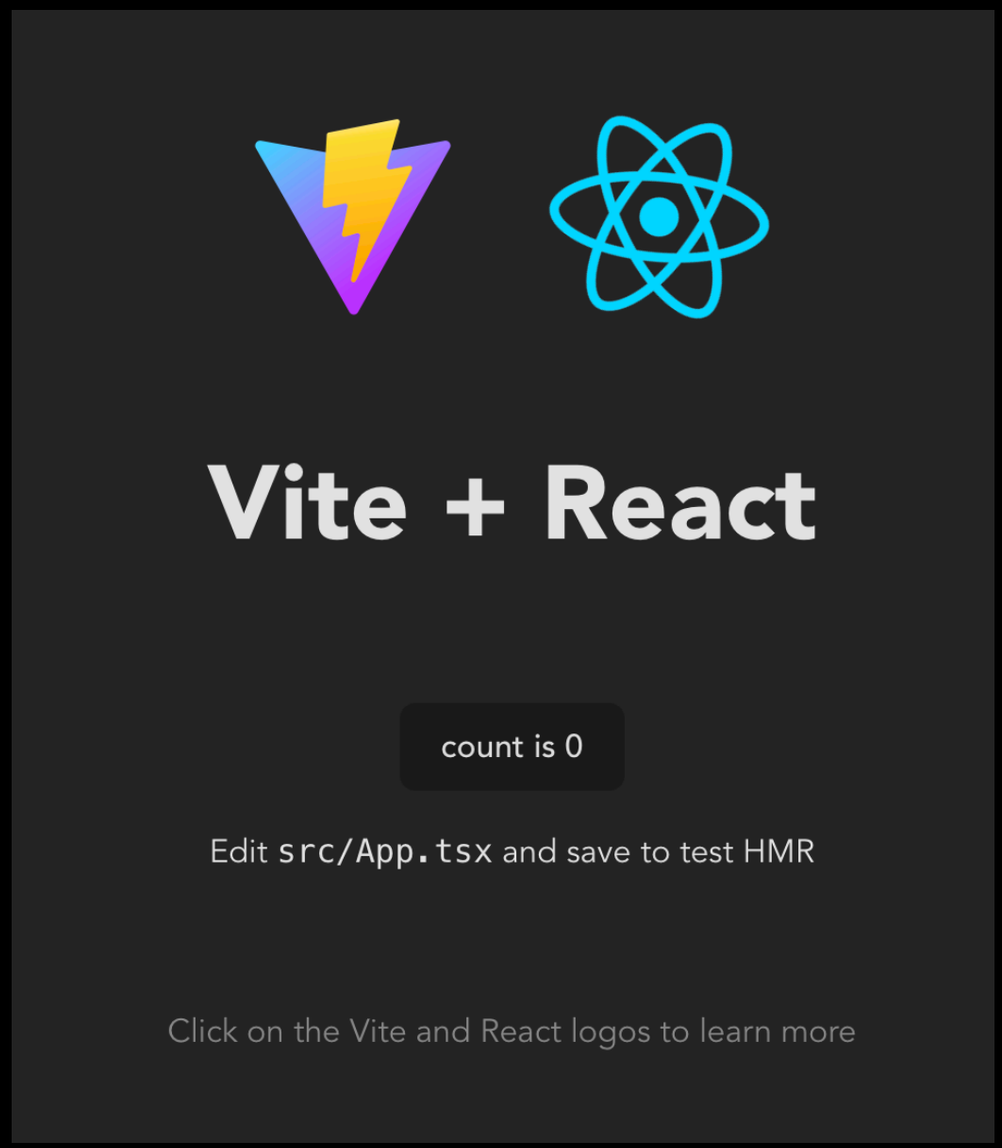
Day 3 - Fundamentals of Frontend Development

```
 ~ npm create vite@latest
✔ Project name: … vite-project
✔ Select a framework: › React
✔ Select a variant: › TypeScript

Scaffolding project in /Users/natalie_agus/vite-project...

Done. Now run:

  cd vite-project
  npm install
  npm run dev
```



Create Component

Set props type, set return type

```tsx
import { ReactElement } from "react";

// setting type for Heading props
type HeadingProps = {
  title: string;
};

// also set return type of Heading which is a JSX element
const Heading = ({ title }: HeadingProps): ReactElement => {
  return <div>{title}</div>;
};

export default Heading;


import Heading from "./components/Heading";

// we don't have to put return ReactElement here because it's expected
function App() {
  return <Heading title={"Happy New Year 2023!"} />;
}

export default App;
```

Default props

React 18.3+ allows inline declaration of default props

```tsx
type SectionProps = {
  title?: string;
  children: ReactNode;
};

export const Section = ({
  children,
  title = "Custom Subheading Default",
}: SectionProps) => {
  return (
    <section>
      <h2>{title}</h2>
      <p>{children}</p>
    </section>
  );
};
```

```tsx
<Heading title={"Happy New Year 2024!"} />
<Section>
  <h3>This is Section's child</h3>
</Section>
```

useState

Define the type of the state used

```tsx
const Counter = () => {
  const [count, setCount] = useState<number>(1);
  const increment = () => {
    setCount((prev) => prev + 1);
  };

  const decrement = () => {
    setCount((prev) => prev - 1);
  };
  return (
    <>
      <h1>Count is {count}</h1>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
    </>
  );
};
```

useState

Define the type of the state used

```ts
interface User {
  id: number;
  username: string;
}
```

```ts
const [users, setUsers] = useState<User[] | null>(null);
```

Some types can’t be inferred from the initial value

Pass down callbacks

Define the props signature

```tsx
type CounterProps = {
  increment: () => void;
  decrement: () => void;
  children: ReactNode;
};
```

```tsx
const Counter = ({ increment, decrement, children }: CounterProps) => {
  return (
    <>
      <h1>{children}</h1>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
    </>
  );
};
```

Children type

Function as children

```tsx
type ChildrenType = {
  children: (num: number) => ReactNode;
};

const CounterUseReducer = ({ children }: ChildrenType) => {
...
  return (
    <>
      <h1>{children(state.count)}</h1>
      <div>


<CounterUseReducer>
  {(num: number) => <>Current Count with useReducer: {num}</>}
</CounterUseReducer>
```

Generic Component

Item type is unknown

```tsx
// generics, we don't know the item type beforehand in this list
interface ListProps<T> {
  items: T[];
  render: (item: T) => ReactNode;
}
// help typescript recognise T is a generic by doing <T extends {}> or < T,>
const List = <T extends {}>({ items, render }: ListProps<T>) => {
  return (
    <ul>
      {items.map((item, i) => (
        <li key={i}>{render(item)}</li>
      ))}
    </ul>
  );
};


<List
  items={["Bird", "Cat", "Dog", "🐻"]}
  render={(item: string) => <span className="gold">{item}</span>}
></List>
```

ReactNode vs ReactElement

- Component return value is always **ReactElement** (expected), so we don't have to declare it.
- A **ReactElement** is an object with a type and props.
  - Either created via `JSX` or `React.createElement`
- A **ReactNode** is a ReactElement, a ReactFragment, a string, a number or an **array** of ReactNodes, or `null`, or `undefined`, or a `boolean`
  - ReactNode is wider, it represents anything React can render.

useEffect

Pretty straightforward

```
useEffect(() => {
  // Runs when the component mounts
  console.log("mount"); // strictmode will mount twice
  console.log("Users: ", users);

  return () => {
    console.log("cleanup unmount");
  };
}, [users]);
```

useCallback

Declare the callback’s input value

```
  // adding event to the callback in React 18 forces you to
implicitly state the type of the event
  // you can just use e: any of course, but that's not neat
  const addTwo = useCallback(
    (e: MouseEvent<HTMLButtonElement> |
KeyboardEvent<HTMLButtonElement>) => {
      setCount((prev) => prev + 2);
    },
    [] // doesn't have to be recreated after it is created the
first time, else fill up the dependency array here
  );
```

useMemo

Straightforward, not necessary to declare memo return type if the function is properly typed

```ts
type fibFunc = (n: number) => number;

// an expensive calculation
const fib: fibFunc = (n) => {
  if (n < 2) return n;
  return fib(n - 1) + fib(n - 2);
};
```

```ts
const fibResult = useMemo<number>(() => fib(someNumber), [someNumber]); ✅
```

```ts
const fibResult = useMemo(() => fib(someNumber), [someNumber]); ✅
```

useRef

Uncontrolled form recap

```tsx
const inputRef = useRef<HTMLInputElement>(null);

console.log(
  "🚀 ~ file: Tutorial.tsx:33 ~ Tutorial ~ inputRef",
  inputRef?.current?.value
);

const handleInputChange = (e: ChangeEvent) => {
  console.log(inputRef?.current?.value);
};
```

```tsx
<input ref={inputRef} type="text" onChange={handleInputChange} />
```

useReducer

Define reducer function state and action types

```ts
const initState = { count: 0 };

// or you can use enum or object
with key:value
const enum REDUCER_ACTION_TYPE {
  INCREMENT,
  DECREMENT,
}

type ReducerAction = {
  type: REDUCER_ACTION_TYPE;
};
```

```ts
const reducer = (
  state: typeof initState,
  action: ReducerAction
): typeof initState => {
  // a large switch statement
  switch (action.type) {
    case REDUCER_ACTION_TYPE.INCREMENT:
      return { ...state, count: state.count + 1 };
    case REDUCER_ACTION_TYPE.DECREMENT:
      return { ...state, count: state.count - 1 };
    default:
      throw new Error();
  }
};
```

useReducer

Define reducer function state and action types

```tsx
const CounterUseReducer = ({ children }: ChildrenType) => {
  const [state, dispatch] = useReducer(reducer, initState);

  const increment = () => {
    dispatch({ type: REDUCER_ACTION_TYPE.INCREMENT });
  };
  const decrement = () => {
    dispatch({ type: REDUCER_ACTION_TYPE.DECREMENT });
  };

  return (
    <>
      <h1>{children(state.count)}</h1>
      <div>
        <button onClick={increment}>+</button>
        <button onClick={decrement}>-</button>
      </div>
    </>
  );
};
```

useReducer

Declaring optional state

```
const initState = { count: 0, message: "" };

const enum REDUCER_ACTION_TYPE {
  INCREMENT,
  DECREMENT,
  NEW_INPUT,
}
```

```
type ReducerAction = {
  type: REDUCER_ACTION_TYPE;
  payload?: string;
};
```

```
 case REDUCER_ACTION_TYPE.NEW_INPUT:
   return { ...state, message: action.payload ?? "" };
```

useReducer

Using the optional state as per normal

```
  const handlePayload = (e: ChangeEvent<HTMLInputElement>) => {
    dispatch({ type: REDUCER_ACTION_TYPE.NEW_INPUT, payload:
e.target.value });
  };
```

```
<br />
<input type="text" onChange={handlePayload} />
<h3>{state.message}</h3>
```

Context

Share data on a global level: create context

This forces context provider to have all default values

```
import React from 'react';

export const AppContext = React.createContext({
  authenticated: true,
  lang: 'en',
  theme: 'dark'
});
```

This allows context provider to skip having default values

```
export const AppContext =
  React.createContext<Partial<ContextProps>>({});
```

Context

Share data on a global level: provide context

```
const App = () => {
  return <AppContext.Provider value={ {
    lang: 'de',
    authenticated: true,
    theme: 'light'
  } }>
    <Header/>
  </AppContext.Provider>
}
```

Context

Share data on a global level: consume context

```
const Header = () => {
  return <AppContext.Consumer>
  {
    ({authenticated}) => {
      if(authenticated) {
        return <h1>Logged in!</h1>
      }
      return <h1>You need to sign in</h1>
    }
  }
  </AppContext.Consumer>
}
```

useContext + useReducer

Create context + custom hook to consume the context, and return only the relevant values

```tsx
import { useCounterMessageHook } from "../hooks/UseCounterMessageHook";
import { useCounterNumberHook } from "../hooks/UseCounterNumberHook";
type ChildrenType = {
  // children is a function that accepts one input which is a number, and returns a react node
  children: (num: number) => ReactNode;
};

const CounterUseContext = ({ children }: ChildrenType) => {
  const { count, increment, decrement } = useCounterNumberHook();
  const { message, handlePayload } = useCounterMessageHook();
  return (
    <>
      <h1>{children(count)}</h1>
      <div>
        <button onClick={increment}>+</button>
        <button onClick={decrement}>-</button>
      </div>
      <br />
      <input type="text" onChange={handlePayload} />
      <h3>{message}</h3>
    </>
  );
};
```