React

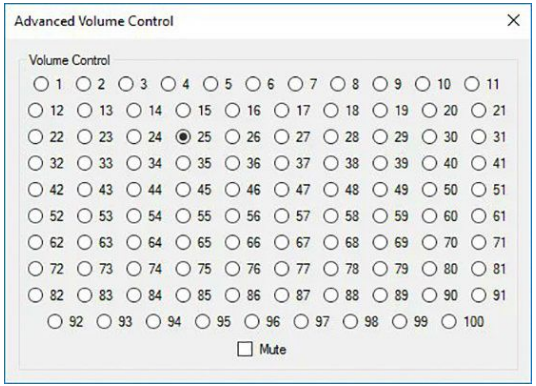
in practice

Outline

- **Introduction**
  What is React.js? - Overview - History
- **Getting Started**
  Installation - Project Structure
- **Key Concepts**
  Components - Rendering - Props - State
- **Hooks**
  useState - useEffect - useContext

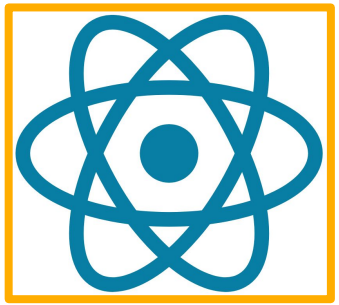
GUI

- GUI (Graphical User Interface) deployment is an hard task
  - https://thebiguglywebsite.com/
- Must be users-proof
  - UX is important: https://userinyerface.com/
- Modern UI are complex → use a framework



Front-End Dev

A lot of libraries (different approach, performance, learning curve, support, ...)

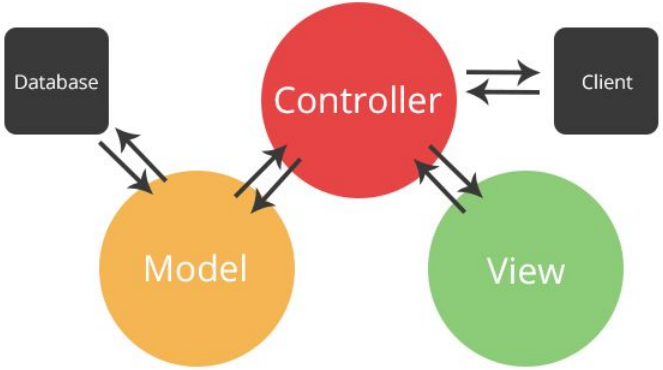


React Overview

- *React* is an *open-source project* created by Facebook in 2013
- React is one of the most popular JavaScript library to build *User Interfaces* (UI). The term *library* is a bit misleading when compared with other library like *jQuery*
- Is not a framework (unlike *Angular*, or *Vue*), but sometimes is referred as a front-end framework

What is React

- React is the V of an *MVC* application (Model View Controller)
- But also the C (depends on the use made of it)



Timeline

- 2011: first deployment in Facebook's feed
- 2012: Instagram use React to build its interface
- 2013: React is Introduced as Open Source (@ JS ConfUS)
- 2015: React stable and established (Netflix and AirBnB endorsement)
  - React Native for IOS and Android
- 2017: React Fiber
- 2019: React Hooks are included in v16.8
- 2020: React v17, Server component and Concurrent mode
- 2022: React v18
- 2023: Create_react_app is dead, lol

React Success

Success given by:

- **Clean programming**: code is very readable and easy to reuse
- **Fast performance**: provides a very quick response time
- **Strong community**: a lot of ready-to-use package for React, someone has already solved your problem

Some of the major companies that currently use React include: Netflix, Facebook, Instagram, Airbnb, Reddit, Dropbox, ...

Install React

- Packages inclusion in HTML header
  - Create an HTML file and using script tag in header to download required packages.
  - NB: It does a slow runtime code transformation, so is only recommended for simple demos.

- Online playgrounds: PlayCode, CodePen, CodeSandbox, o Stackblitz

- **Local installation using a framework: Next.js**

Prerequisites

- Basic knowledge of **DOM, HTML, CSS, JavaScript and TypeScript**
- **Node.js** (version ≥ 18)
- Integrated Development Environment (IDE) such as **Visual Studio Code**

https://www.typescriptlang.org/
https://nodejs.org/en
https://code.visualstudio.com/

TypeScript

"TypeScript is **JavaScript with syntax for types**"

```
interface Account {
  id: number
  displayName?: string
  version: 1 | 2
}

function welcome(user: Account) {
  console.log(user.id)
}
```

```
type Result = "pass" | "fail"

function verify(result: Result) {
  if (result === "pass") {
    console.log("Passed")
  } else {
    console.log("Failed")
  }
}
```

The JavaScript language: https://javascript.info/js

JavaScript Reference: https://developer.mozilla.org/en-US/docs/Web/JavaScript

The TypeScript Handbook: https://www.typescriptlang.org/docs/handbook/intro.html

Installation

The recommend way for starting a new Next.js app is using [create-next-app](), which sets up everything automatically. To create a project, run:

```
npx create-next-app@latest
```

On installation, you'll see the following prompts:

```
What is your project named?  my-app
Would you like to use TypeScript?  No / Yes
Would you like to use ESLint?  No / Yes
Would you like to use Tailwind CSS?  No / Yes
Would you like to use `src/` directory?  No / Yes
Would you like to use App Router? (recommended)  No / Yes
Would you like to use Turbopack for `next dev`?  No / Yes
Would you like to customize the default import alias (@/*)?  No / Yes
What import alias would you like configured?  @/*
```

Available Scripts

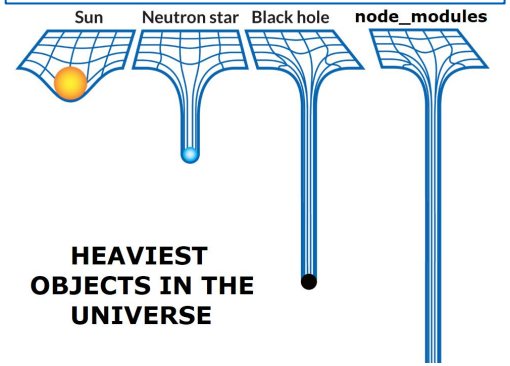
- `npm run dev`
  Start the application in **development mode** with hot-code reloading, error reporting, ...
  The application will start at `http://localhost:3000` by default. The default port can be changed with `-p`
  You can also set the hostname to be different from the default of `0.0.0.0`, this can be useful for making the application available for other devices on the network. The default hostname can be changed with `-H`
- `npm run build`
  Create an **optimized production build** of your application.
- `npm start`
  Start the application in **production mode**.
- `npm lint`
  Run ESLint for all files in the `pages/`, `app/`, `components/`, `lib/`, and `src/` directories.

Project Structure

- my-app
  - .next
  - app
    - fonts
    - favicon.ico
    - globals.css
    - layout.tsx
    - page.tsx
  - node_modules
  - public
  - .eslintrc.json
  - .gitignore
  - next-env.d.ts
  - next.config.ts
  - package-lock.json
  - package.json
  - postcss.config.mjs
  - README.md
  - tailwind.config.ts
  - tsconfig.json

App Router

The folder where npm (or Yarn) installs all the **dependencies**



Static assets (images, fonts, ...) to be served

Git files and folders to ignore

Configuration file for Next.js

Project dependencies and scripts

Configuration file for TypeScript

Component-Based Architecture

- GUI as a group of *components*
- Increase:
  - Modularity & Reusability
  - Consistency & Scalability
  - Faster Development & Maintenance
  - Enhanced Collaboration
  - Testing & Adaptability: Simplifies

JSX

The simplest example of React Application:

```
import ReactDOM from "react-dom/client"; //using CodeSandBox
const container = document.getElementById('root');
const root = ReactDOM.createRoot(container);

root.render(<h1>Hello, world!</h1>);
```

React 18

JSX

- JSX (JavaScript + XML) is a widely used extension of JS
- *Markup code* and *logic* are included together
- it is ***syntactic sugar*** that allow to insert JS in HTML code and to produce **React elements**
- Using JSX *is not mandatory* for writing React, but it is widely appreciated. Under the hood, it's running *createElement*, which takes tags, properties, childrens and renders the same information.

JSX translation: https://babeljs.io/repl/
HTML to JSX: https://transform.tools/html-to-jsx

JSX

- JSX is closer to JavaScript than HTML, so there are a few *key differences* to note when writing it:
  - className is used instead of *class* for adding CSS classes, as class is a *reserved keyword* in JavaScript
  - Properties and methods in JSX are camelCase - e.g., *onclick* will become *onClick*
  - Self-closing (no children or content) tags should end in a slash - e.g. <img />
  - ...

JSX

- JavaScript expressions can also be embedded inside JSX using *curly braces* { }, including *variables*, *functions*, and *properties*

```
import ReactDOM from "react-dom/client";
let btnLabel = "I'm a button"

function dt() {
   date = new Date()
   return "Hello. it's " + date.toLocaleTimeString()
}

const root = ReactDOM.createRoot(
  document.getElementById('root')
);
root.render(
       <button onClick={() => console.log(dt())}>
       Hello, {btnLabel} </button>
);
```

React Element - Rendering

- Renderization is the operation that take React Elements and return DOM tree (***render*** method)
- Usually, the DOM tree of a React app is placed under a *single root node*

```
<div id="root"></div>
```

→

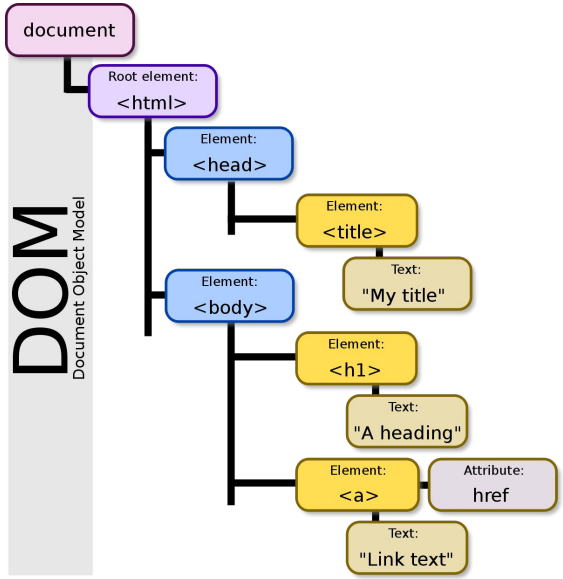
```
const root = ReactDOM.createRoot(
document.getElementById('root'));
const element = <h1>Hello, world</h1>;
root.render(element);
```

- React does not update all nodes of the DOM tree, but *only those that are modified*, through a mechanism called Reconciliation

React Element - Rendering

React use the **Virtual DOM** (VDOM) paradigm, and use declarative API to define element changes

- *DOM*: programming interface that represents an HTML document (e.g., web pages) as a tree-like structure. It's created by the browser
- *VDOM*: abstraction of the real DOM, created and maintained by JavaScript libraries such as React. It is lightweight and efficient.



React Element - Rendering

- Reconciliation is done using an O(n) diff algorithm, thanks to some Heuristic rules:
  - *Different component types are assumed to generate substantially different subtrees. React will not attempt to diff them, but rather replace the old tree completely.*
  - *Diffing of lists is performed using keys prop, implemented by developers.*

- It is a diff algorithm, so every element is checked between the previous e the new DOM

React Element - Rendering

OLD

```
<div>
  <Counter />
</div>
```

diff

NEW

```
<span>
  <Counter />
</span>
```

All the subtree is updated

OLD

```
<div className="A" title="stuff" />
```

diff

NEW

```
<div className="B" title="stuff" />
```

Only classname is updated

OLD

```
<div style={{color: 'red',
            fontWeight: 'bold'}} />
```

diff

NEW

```
<div style={{color: 'blue',
            fontWeight: 'bold'}} />
```

Only color is updated

React Element - Rendering

OLD

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>
```

diff

NEW

```
<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

→ Only “third” is updated

OLD

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>
```

diff

NEW

```
<ul>
  <li>third</li>
  <li>first</li>
  <li>second</li>
</ul>
```

→ All the <li> element are updated (and also their subtrees)

React Element - Rendering

- Using the Key prop to we could update list elements matching their ID

OLD

```
<ul>
  <li key="1">first</li>
  <li key="2">second</li>
</ul>
```

diff

NEW

```
<ul>
  <li key="3">third</li>
  <li key="1">first</li>
  <li key="2">second</li>
</ul>
```

→ Only “third” is updated

- Keys should be “*stable, predictable, and unique*”

Components

React apps are made out of **components**. A component is a piece of the UI (user interface) that has its own logic and appearance. A component can be as small as a button, or as large as an entire page.

```
const MyButton = () => <button>I'm a button</button>
export default MyButton
```

React component names must always **start with a capital letter**, while HTML tags must be lowercase.

A component must be **pure**, meaning: it should not change any objects or variables that existed before rendering and, given the same inputs, a component should always return the same JSX.

https://react.dev/learn/your-first-component

React Component

- Components are the main way to create elements.
  They are *JavaScript functions* or *classes* that return DOM elements.
  NB: *Classes are no more used*.

- Components let you split the UI into *independent*, *reusable* pieces, and think about each piece in *isolation*

```
function Greet (props) {
 return <h1>Hello, {props.nome}</h1>;
}
```

```
class Greet extends React.Component {
 render() {
   return <h1>Hello, {this.props.nome}</h1>;
 }
}
```

React Component

- Components can reference one or more other components in their output, using nesting. *Component identification* is a core activity in React development
- Components starts with a *capitalized* letter
- ***export default*** keywords specify the main component in the file
- A file can include many components, or a component could be content into a single file

```
function MyButton() {
  return (
      <button>
      I'm a button
      </button>
  );
}

function MyApp() {
  return (
      <div>
      <h1>Welcome to my app</h1>
      <MyButton />
      </div>
  );
}

const root = ReactDOM.createRoot(
document.getElementById('root'));
root.render(<MyApp />);
```

Props

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "**props**") and return React elements describing what should appear on the screen.

```
type UserThumbnailProps = {
  img: string,
  url: string,
}

export const UserThumbnail = (props: UserThumbnailProps) => (
  <a href={props.url}><img src={props.img} /></a>
)
```

Props are read-only: all React components must act like pure functions with respect to their props.

https://react.dev/learn/passing-props-to-a-component

Component Props

- Components could use function parameters to specialize its behaviour
- React components use PROPS to communicate with each other

```
import ReactDOM from "react-dom/client"; //using CodeSandBox

function Greet(props){
  const nome = props.name;
  return ( <h1> Hello, {nome} </h1> );
}
function MyApp(){
  return (
      <div>
      <Greet name="Mario" />
      <Greet name="Sara" />
      <Greet name="Paola" />
      </div>
  );
}

const root = ReactDOM.createRoot( document.getElementById('root'));
root.render(<MyApp />);
```

Component Props

- We can pass properties to the element *during element creation*.
  A component can access his properties using the keyword props

- Props can be any JavaScript: objects, arrays, functions, and even JSX (≠ HTML attributes)

- The props are read-only: a method must never modify them - so we want only method as ***Pure Function*** -> *idempotent and isolated*

- Props flows in one way: *from parent to children*

Component Props

Example using props

→ **Conditional Rendering**:

> components will display different things depending on different conditions

```
import ReactDOM from "react-dom/client";  //using CodeSandBox
const root = ReactDOM.createRoot( document.getElementById('root'));

function Item({ listname, isPresent }) {
  return (
    <li className="item"> {listname} {isPresent && '- OK' ||  "- Da
comprare"}
</li>
  );}

function ElementList() {
  return (
    <section>
      <h1>Lista Spesa</h1>
      <ul>
        <Item    isPresent={true}    listname="Latte"/>
        <Item    isPresent={true}    listname="Pane"/>
        <Item    isPresent={false}   listname="Frutta"/>
      </ul>
    </section>
  );
}
root.render(ElementList())
```

Conditional Rendering

```
let content;
if (isLoggedIn) {
  content = <AdminPanel />;
} else {
  content = <LoginForm />;
}
return (
  <div>
    {content}
  </div>
);
```

```
<div>
  {isLoggedIn ? (
    <AdminPanel />
  ) : (
    <LoginForm />
  )}
</div>
```

```
<div>
  {isLoggedIn && <AdminPanel />}
</div>
```

https://react.dev/learn/conditional-rendering

Arrow Function Expressions

```
(param) => expression  // The parentheses are optional with one single parameter
(param_1, ..., param_N) => expression
```

An **arrow function expression** is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

```
const func = (function (a) {
  return a + 100;
});
```

```
const func = a => a + 100;
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

List and Keys

```
const products = [
  { title: 'Cabbage', id: 1 },
  { title: 'Garlic', id: 2 },
  { title: 'Apple', id: 3 },
];

export const ShpoppingList = () => (
  <ul>
    {products.map(p => <li key={p.id}>{p.title}</li>)}
  </ul>
)
```

Keys help React identify which items have changed, are added, or are removed.
Keys should be given to the elements inside the array to give the elements a **stable identity**.

https://react.dev/learn/rendering-lists

Handling Events

You can respond to events by declaring **event handler** functions inside your components:

```
const MyButton = () => {
    const handleClick = () => {
        alert("You have clicked the button")
    }
    return <button onClick={handleClick}>Click Here!</button>
}
```

Do not call the event handler function: you only need to **pass it down**.

https://react.dev/learn/responding-to-events

Component State

- The state (or status) is similar to props, but it is private and *completely controlled* by the component

- Think about state as any data that should be saved and modified *without necessarily* being added to a *database* - for example, *shopping cart* before confirming your purchase

- State is associated with *Rendering* and *Lifecycle* (more later) → **interactivity**

Component State

- State was born for classes:
  - state is the keyword to access them
  - setState the method to update (outside the constructor)

…. But classes are no more used :(

<u>So how to use State with Component Functions?</u>

```
import ReactDOM from "react-dom";//using CodeSandBox
const root = ReactDOM.createRoot(
        document.getElementById('root'));
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
}
  render() {
    return (
      <div>
       <h1>Hello, world!</h1>
       <h2>It is
         {this.state.date.toLocaleTimeString()}.</h2>
      </div>
   );
}}

root.render(<Clock />);
```

never forget where you came from

React Hook

- Hooks were introduced with React version 16.8
- Briefly, they are a way to *use React functionality in any place*
- From a practical perspective, React Hooks are simple *JavaScript functions* that we can use to isolate the reusable part from a functional component.
- Hooks can be stateful and manage side-effects, so function Components could be stateful.
- Note: hooks function start with "*use*"

React Hook

- For example, with the useState Hook you can also use it in *Functions*, making the function component *stateful* as well.
- Normally variables disappear after a function end, but React will preserve this state variables
- useState accept an *argument* (initial state) and returns a *pair* (current state value and a update function)

```
import ReactDOM from "react-dom/client"; //using CodeSandBox
import React, { useState } from 'react';
function Counter() {
   const [contatore, setContatore] = useState(0);
   return (
   <div>
       <p>Hai cliccato {contatore} volte</p>
       <button onClick={() => setContatore(contatore + 1)}>
           Click me
       </button>
   </div>
   );
}
const root = ReactDOM.createRoot(
document.getElementById('root'));
root.render(<Counter />);
```

React Hook

State Creation

```
function Counter() {
   const [contatore, setContatore] = useState(0);
   const [contatore2, setContatore2] = useState(0);
...
```

State Read

```
<div>
       <p>Hai cliccato {contatore} volte</p>
...
```

State Update

```
<button onClick={() => setContatore(contatore + 1)}>
      Click me
</button>
...
```

React Hook

NB: Setting a state does not change the state variable you already have, but instead triggers a re-render.

→ State as a *Snapshot*: is fixed until next render!

Example: counter increments 3 times

```
import ReactDOM from "react-dom/client";
import { useState } from 'react';

function Counter() {
  const [number, setNumber] = useState(0);

  return (
        <div>
        <h1>{number}</h1>
        <button onClick={() => {
        setNumber(number + 1);
        setNumber(number + 1);
        setNumber(number + 1);
        }}>+3</button>
        </div>
  )
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Counter />)
```

React Hook

NB: Do not mutate State directly in React (anti-pattern)
State mutation must be done using the setState method, given in output by useState hook
→ Rendering is triggered

- There are a few reasons, like Debugging, Optimization strategies and adherence with react new feature
- State could be any object: it is possible to use the *Spread syntax* to create shallow copies of the object and modify only the changed values

```
const [person, setPerson] =
useState({
        firstName: "Samus",
        lastName: "Aran",
        email: "samaran@metroid.jp"
  });
```

```
setPerson({
  firstName: "New Name",
  lastName:
person.lastName,
  email: person.email
});
```

→

```
setPerson({
  ...person,
  firstName: "New Name"
});
```

Managing the State

```
export const MyButton = () => {
  const [count, setCount] = useState(0)

  function handleClick() {
    setCount(prev => prev + 1)
  }
  return <button onClick={handleClick}>Clicked {count} times</button>
}
```

- Do not modify the state **directly**
- State updates may be **asynchronous**
- State updates are **merged**

https://reactjs.org/docs/state-and-lifecycle.html

Props vs State

There are two types of “model” data in React: props and state. The two are very different:

- **Props are like arguments you pass to a function.** They let a parent component pass data to a child component and customize its appearance.
  For example, a `Form` can pass a `color` prop to a `Button`.
- **State is like a component’s memory.** It lets a component keep track of some information and change it in response to interactions.
  For example, a `Button` might keep track of `isHovered` state.

Props and state are different, but they work together. A parent component will often keep some information in state (so that it can change it), and *pass it down* to child components as their props.

https://react.dev/learn/thinking-in-react#step-4-identify-where-your-state-should-live

Hooks and Function Components

Hooks (React ≥ 16.8) let you use React features without writing a class.

Rules of Hooks:

1. Only call Hooks **at the top level**. Don't call Hooks inside loops, conditions, or nested functions.
2. Only call Hooks **from React function components** (and custom hooks). Don't call Hooks from regular JavaScript functions.

In JavaScript, all functions work like closures. A **closure** is a function, which uses the scope in which it was declared when invoked (not the scope in which it was invoked).

https://react.dev/learn/state-a-components-memory

Hooks

- **useState**
- **useEffect**
- **useContext**
- useReducer
- useCallback
- useMemo
- useRef
- ...

```
const [state, setState] = useState(initialState);
```

```
useEffect(() => {
  // This run after the render is committed to the screen
  return () => {
    // This run when the component leaves the screen
  }
}, [
  // Array of values that the effect depends on
  // Leave it empty if you want to run the effect only once
])
```

https://react.dev/reference/react/hooks

useEffect

The *Effect Hook* lets you perform **side effects** (e.g., data fetching, setting up a subscription, and manually changing the DOM) in function components.

```
useEffect(
  () => {
    // Execute the side effects
    return () => {
      // Clean up the previous effect before executing the next effect
    }
  }, [
    // Fire the effect only when the values listed here have changed
    // If empty, the effect is run and clean up only once (on mount and unmount)
  ]
)
```

https://reactjs.org/docs/hooks-effect.html

Context

Context provides a way to **pass data through the component tree** without having to pass props down manually at every level.

```
const MyContext = React.createContext(defaultValue)

const App = () => (
  <MyContext.Provider value={/* some value */}>
    {/* ...child components... */}
  </MyContext.Provider>
)
```

```
const MyComponent = () => {
  const value = useContext(MyContext)

  return <span>The value of MyContext is {value}</span>
}
```

Use cases: theming, current account, routing, managing state, ...

https://reactjs.org/docs/context.html

useRef

Essentially, `useRef` is like a “box” that can hold a mutable value in its `.current` property.

```
const Form = () => {
    const [name, setName] = useState('')
    const handleSubmit: React.FormEventHandler = e => {
        e.preventDefault()
        alert("Your name is " + name)
    }
    return (
        <form onSubmit={handleSubmit}>
            <input onChange={e => setName(e.target.value)} value={name} />
            <button type="submit">Submit</button>
        </form>
    )
}
```

Controlled Component

```
const Form = () => {
    const ref = useRef<HTMLInputElement>(null)
    const handleSubmit: React.FormEventHandler = e => {
        e.preventDefault()
        alert("Your name is " + ref.current?.value)
    }
    return (
        <form onSubmit={handleSubmit}>
            <input ref={ref} />
            <button type="submit">Submit</button>
        </form>
    )
}
```

Uncontrolled Component

vs

https://reactjs.org/docs/hooks-reference.html#useref
https://goshacmd.com/controlled-vs-uncontrolled-inputs-react/

Error Boundaries

Error boundaries (React ≥ 16) are components that **catch JavaScript errors anywhere in their child component tree**, log those errors, and display a fallback UI instead of the component tree that crashed.

```
type ErrorBoundaryProps = { children?: ReactNode }
type ErrorBoundaryState = { hasError: boolean }

class ErrorBoundary extends Component<ErrorBoundaryProps, ErrorBoundaryState> {
  public state: State = { hasError: false }

  public static getDerivedStateFromError(_: Error): State {
    return { hasError: true } // Update state so the next render will show the fallback UI.
  }
  public componentDidCatch(error: Error, errorInfo: ErrorInfo) {
    console.error("Uncaught error:", error, errorInfo);
  }
  public render() {
    return this.state.hasError ? (<h1>Sorry.. there was an error</h1>) : this.props.children
  }
}
```

https://reactjs.org/docs/error-boundaries.html

NEXT.JS

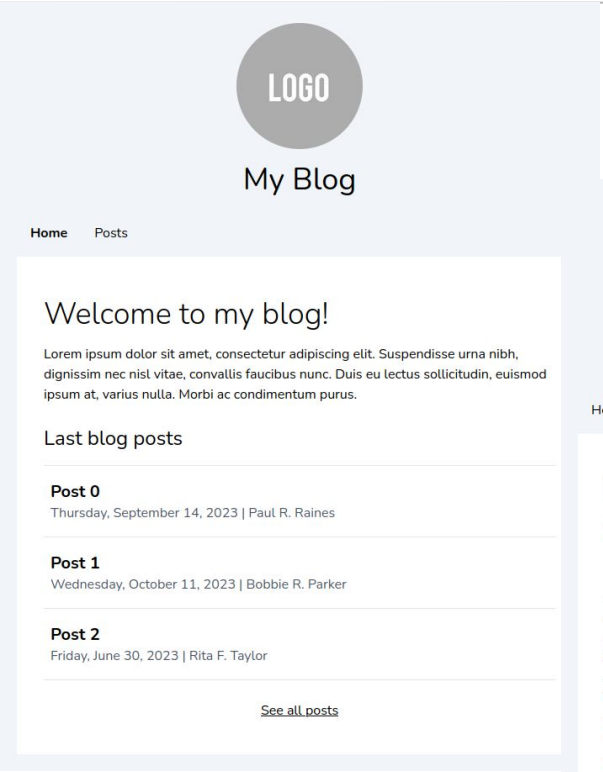
in practice

Outline

- **Basic Concepts of Next.js**
  What is Next.js? - Optimizations - Rendering
- **Getting Started**
  Installation - Available scripts - Project Structure
- **Building an Application**
  Routing - Rendering - Data fetching - Styling
- **Dependencies**
- **Conclusions**

Today's Goals

- Understand base features of Next.js
- Learn the difference between Client and Server Components
- Discover best practices for Next.js applications
- Implement a very simple blog web application



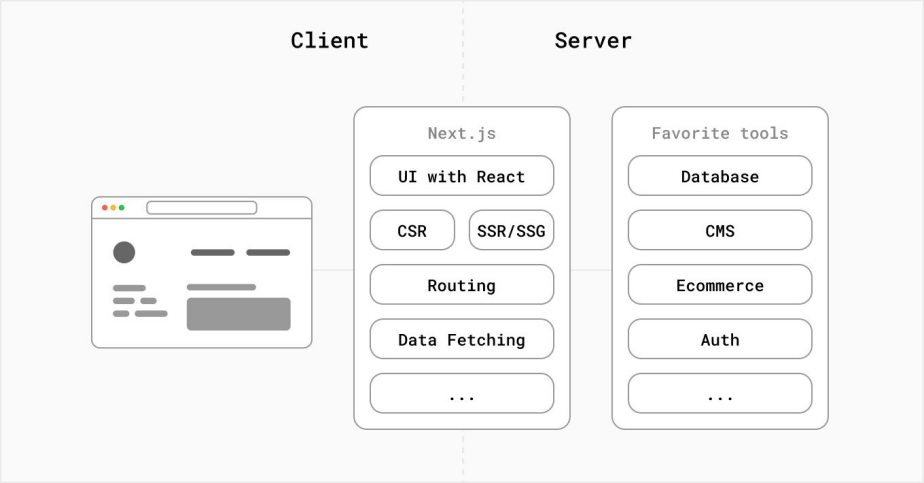
https://github.com/franksacco/react-in-practice

What is Next.js?

*Next.js is a React **framework** that gives you building blocks to create web applications.*
By framework, we mean Next.js handles the tooling and configuration needed for React, and provides additional structure, features, and optimizations for your application.

Main features of Next.js:

- routing (file-system based router)
- rendering (client-side and server-side rendering)
- data fetching
- styling
- optimizations



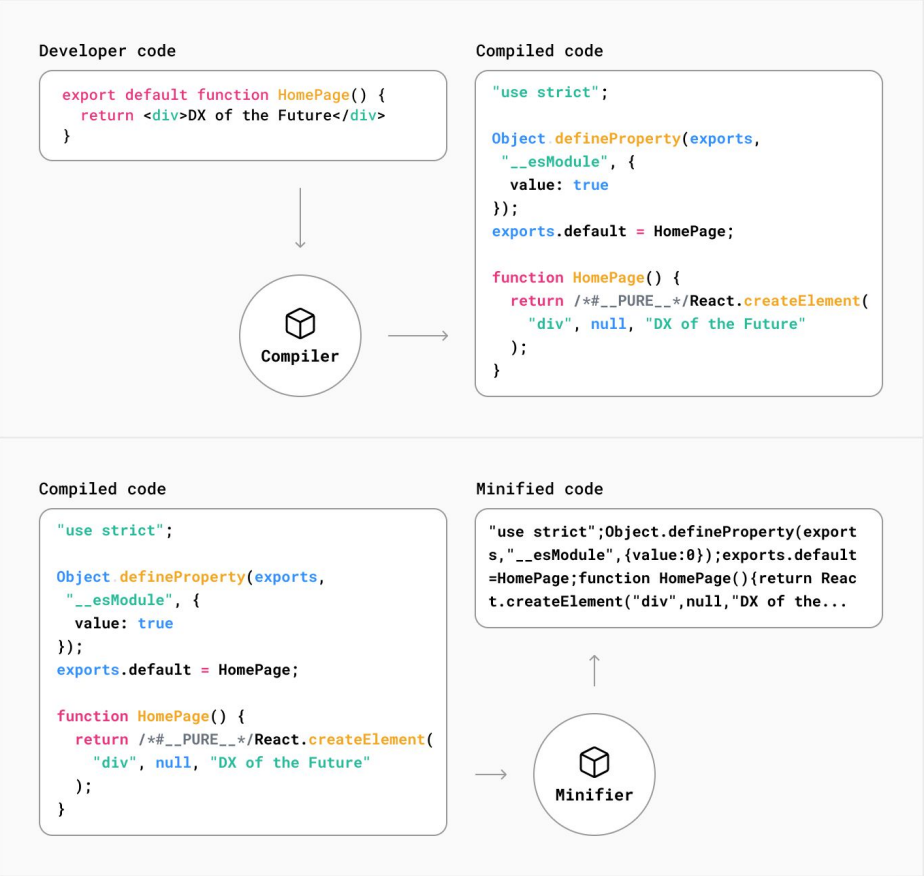
Optimizations (1/2)

Compiling

Developers write code in *developer-friendly* languages such as JSX, TypeScript, and modern versions of JavaScript.
While these languages improve the efficiency and confidence of developers, they need to be **compiled into JavaScript** before browsers can understand them.

Minifying

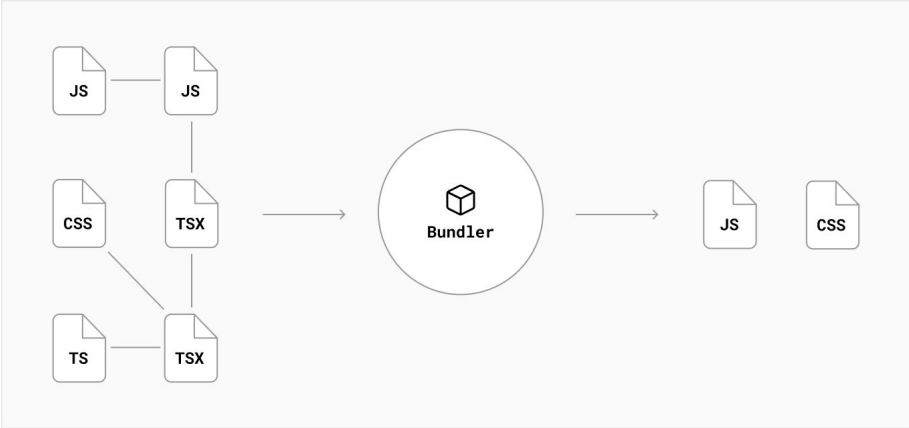
Minification is the process of **removing unnecessary code** formatting and comments without changing the code's functionality.
The goal is to improve the application's performance by decreasing file sizes.



Optimizations (2/2)

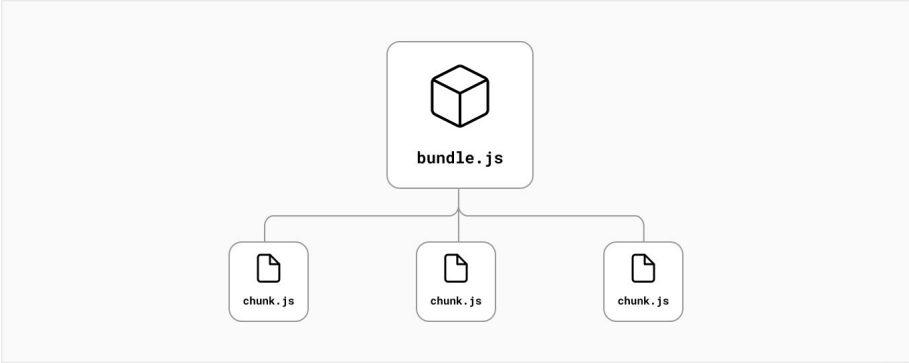
Bundling

Bundling is the process of resolving the web of dependencies and **merging (or 'packaging') the files** (or modules) into optimized bundles for the browser, with the goal of reducing the number of requests for files when a user visits a web page.



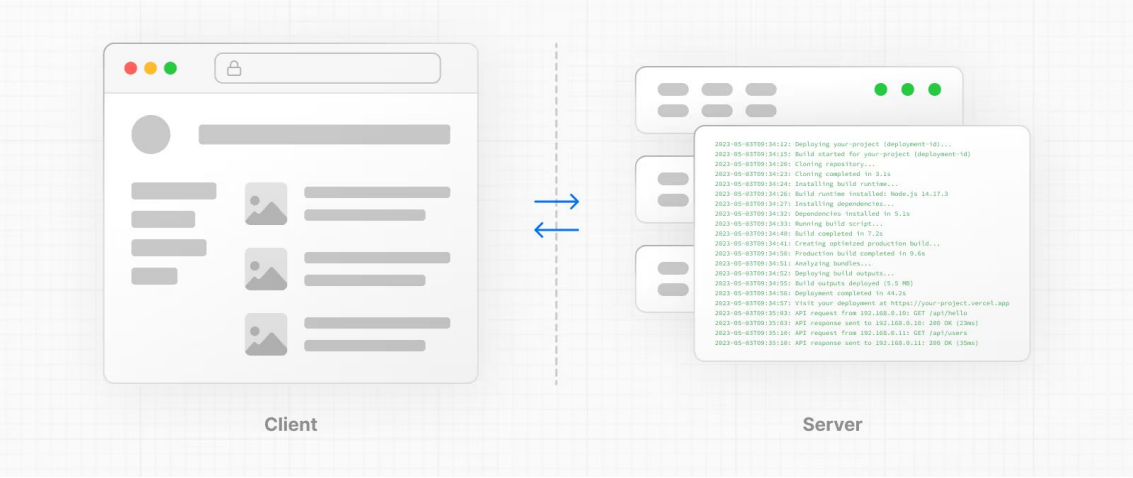
Code-splitting

Code-splitting is the process of **splitting the application's bundle into smaller chunks** required by each entry point. The goal is to improve the application's initial load time by only loading the code required to run that page.



Rendering

Rendering converts the code you write into user interfaces.
Next.js allows you to create hybrid web applications where parts of your code can be rendered on the server or the client.



The **client** sends a request to a server and then turns the response into a user interface.

The **server** stores your application code, receives requests from a client, and sends back an appropriate response

Rendering

- **Server Components** allow you to write UI that can be rendered and optionally cached on the server. Three server rendering strategies:
  - **static rendering** (build time or background)
  - **dynamic rendering** (request time)
  - **streaming** (progressive render of UI)
- **Client Components** allow you to write interactive UI that can be rendered on the client at request time.

By default, Next.js uses Server Components.
You have to explicitly decide what components React should render on the client

Benefits of Server Components:
- Data Fetching
- Security
- Caching
- Bundle Sizes
- Initial Page Load and First Contentful Paint (FCP)
- Streaming

Benefits of Client Components:
- Interactivity
- Browser APIs

"`use client`" is used to declare a boundary between Server and Client Component modules.

When to use Server and Client Components?

What do you need to do?	Server Component	Client Component
Fetch data	✅	❌
Access backend resources (directly)	✅	❌
Keep sensitive information on the server (access tokens, API keys, etc)	✅	❌
Keep large dependencies on the server / Reduce client-side JavaScript	✅	❌
Add interactivity and event listeners (`onClick()`, `onChange()`, etc)	❌	✅
Use State and Lifecycle Effects (`useState()`, `useReducer()`, `useEffect()`, etc)	❌	✅
Use browser-only APIs	❌	✅
Use custom hooks that depend on state, effects, or browser-only APIs	❌	✅

https://nextjs.org/docs/app/building-your-application/rendering/composition-patterns

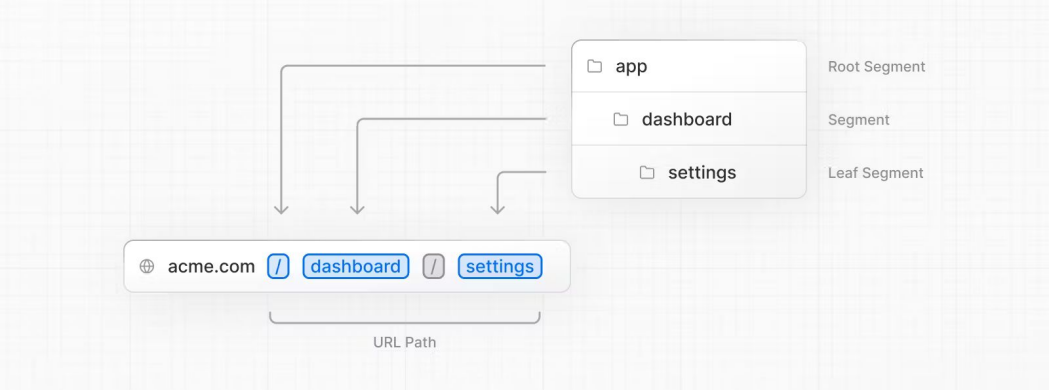
Defining Routes

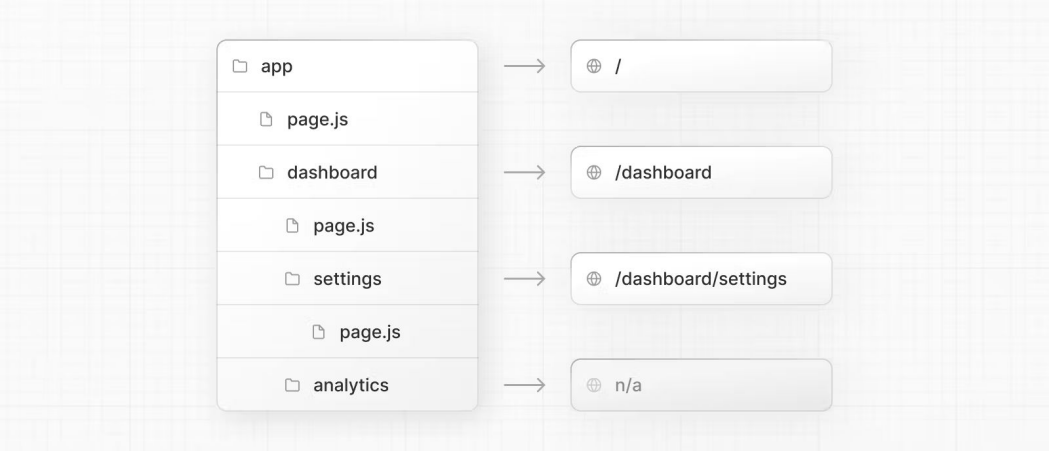
Next.js uses a **file-system based router** where **folders** are used to define routes.

Each folder represents a **route** segment that maps to a **URL** segment.

To create a nested route, you can nest folders inside each other.

A special `page.js` file is used to make route segments publicly accessible.

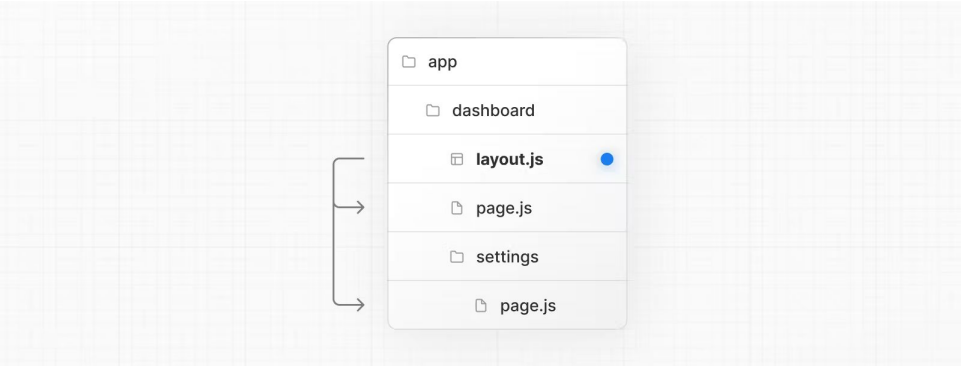




Pages and Layouts

A page is UI that is **unique** to a route.
You can define pages by exporting a component from a `page.js` file.
Use nested folders to define a route and a `page.js` file to make the route publicly accessible.

A layout is UI that is **shared** between multiple pages. On navigation, layouts preserve state, remain interactive, and do not re-render.
Layouts can also be nested.
The **root layout**, that is defined at the top level of the `app` directory and applies to all routes, is **required**.



Linking and Navigating

`<Link>` is a built-in component that extends the HTML `<a>` tag to provide prefetching and client-side navigation between routes. It is the primary way to navigate between routes in Next.js.

```
import Link from 'next/link'
export default function Page() {
  return <Link href="/dashboard">Dashboard</Link>
}
```

You can use `usePathname()` in client components to determine if a link is active.

```
'use client'
import { usePathname } from 'next/navigation'
export default function Nav() {
  const pathname = usePathname()
  return (...)
}
```

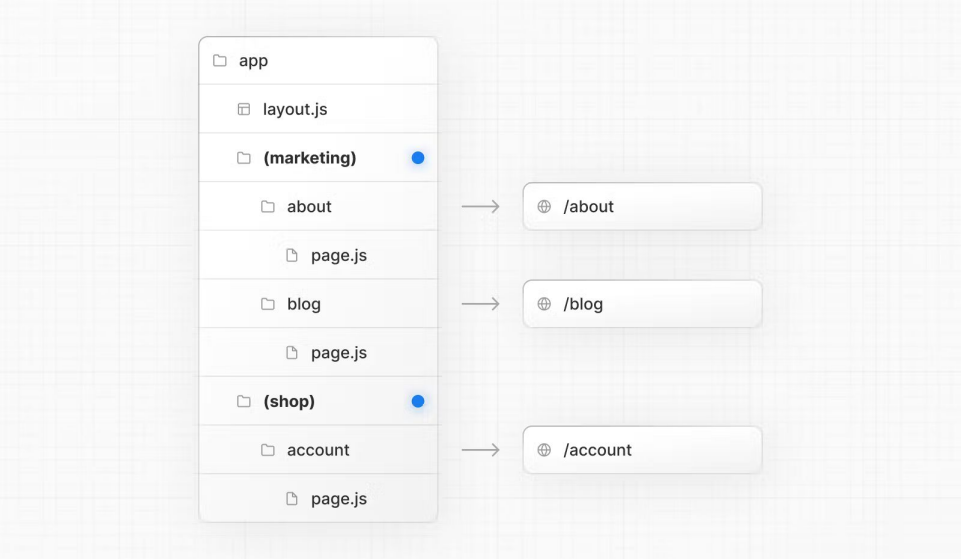
Route Groups

In the app directory, nested folders are normally mapped to URL paths. However, you can mark a folder as a **Route Group** to prevent the folder from being included in the route's URL path.

This allows you to organize your route segments and project files into logical groups without affecting the URL path structure.

Route groups are useful for:

- **organizing routes** into groups e.g. by site section, intent, or team;
- enabling **nested layouts** in the same route segment level.



Dynamic Routes

When you want to create routes from **dynamic data**, you can use Dynamic Segments that are filled in at **request time** or **pre-rendered at build time**.

A Dynamic Segment can be created by wrapping a folder's name in square brackets: **`[folderName]`**. For example, `[id]` or `[slug]`.
Dynamic Segments are passed as the **`params`** prop to `layout`, `page`, `route`, and `generateMetadata` functions.

The **`generateStaticParams`** function can be used in combination with dynamic route segments to **statically generate** routes at build time instead of on-demand at request time.

`[folder]`	Dynamic route segment
`[...folder]`	Catch-all route segment
`[[...folder]]`	Optional catch-all route segment

Routing File Conventions

layout	`.js` `.jsx` `.tsx`	A **root layout** is the top-most layout in the root `app` directory. It is used to define the `<html>` and `<body>` tags and other globally shared UI.
page	`.js` `.jsx` `.tsx`	A **page** is UI that is unique to a route.
loading	`.js` `.jsx` `.tsx`	A **loading** file can create instant loading states built on Suspense.
not-found	`.js` `.jsx` `.tsx`	The **not-found** file is used to render UI when the `notFound` function is thrown within a route segment.
error	`.js` `.jsx` `.tsx`	An **error** file defines an error UI boundary for a route segment.
global-error	`.js` `.jsx` `.tsx`	To specifically handle errors in root `layout.js`, use a variation of `error.js` called `app/global-error.js` located in the root `app` directory.
route	`.js` `.ts`	Route Handlers allow you to create custom request handlers for a given route using the Web `Request` and `Response` APIs.
template	`.js` `.jsx` `.tsx`	A **template** file is similar to a layout in that it wraps each child layout or page. Unlike layouts that persist across routes and maintain state, templates create a new instance for each of their children on navigation.
default	`.js` `.jsx` `.tsx`	Parallel route fallback page.

Data Fetching

1. **On the server, with `fetch`:** Next.js extends the native **`fetch` Web API** to allow you to configure the caching and revalidating behavior for each fetch request on the server.
   React extends `fetch` to automatically memoize fetch requests while rendering a React component tree.

2. **On the server, with third-party libraries** that doesn't support or expose `fetch` (for example, a database, CMS, or ORM client), you can configure the caching and revalidating behavior of those requests using the Route Segment Config Option and React's `cache` function.

3. **On the client, via a Route Handler:** if you need to fetch data in a client component, you can call a Route Handler from the client.

4. **On the client, with third-party libraries:** you can also fetch data on the client using a third-party library such as SWR or React Query. These libraries provide their own APIs for memoizing requests, caching, revalidating, and mutating data.

https://nextjs.org/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating

Styling

Next.js supports different ways of styling your application, including:

- **Global CSS**: Simple to use and familiar for those experienced with traditional CSS, but can lead to larger CSS bundles and difficulty managing styles as the application grows.
- **CSS Modules**: Create locally scoped CSS classes to avoid naming conflicts and improve maintainability.
- **Tailwind CSS**: A utility-first CSS framework that allows for rapid custom designs by composing utility classes.
- **Sass**: A popular CSS preprocessor that extends CSS with features like variables, nested rules, and mixins.
- **CSS-in-JS**: Embed CSS directly in your JavaScript components, enabling dynamic and scoped styling.

https://nextjs.org/docs/app/building-your-application/styling

Some Useful Functions

- **`notFound()`**
  Invoking the `notFound()` function throws a `NEXT_NOT_FOUND` error and terminates rendering of the route segment in which it was thrown.
  The **Not Found UI** defined for the route segment is rendered instead.
- **`redirect(path, type)`**
  The `redirect` function allows you to **redirect the user to another URL**. `redirect` can be used in Server Components, Client Components, Route Handlers, and Server Actions.
- **`usePathname()`**
  `usePathname` is a **Client Component** hook that lets you read the current URL's **pathname**.

https://nextjs.org/docs/app/api-reference/functions

Install a dependency

You can install a package using:

```
npm install <package>
# Alternatively you may use yarn:
yarn add <package>
```

Your package manager will update the lock and `package.json` files accordingly.

The npm registry is a **public collection of packages** of open-source code for Node.js, front-end web apps and the JavaScript community at large.

https://docs.npmjs.com/cli/v8/commands/npm-install

https://www.npmjs.com/

Material UI

Material UI is a library of React UI components that implements Google's **Material Design**.

It includes a comprehensive **collection of prebuilt components** that are ready for use in production right out of the box.

```
npm install @mui/material @emotion/react @emotion/styled
# Robot Font
npm install @fontsource/roboto
# Material Icons
npm install @mui/icons-material
```

https://mui.com/material-ui/getting-started/installation/

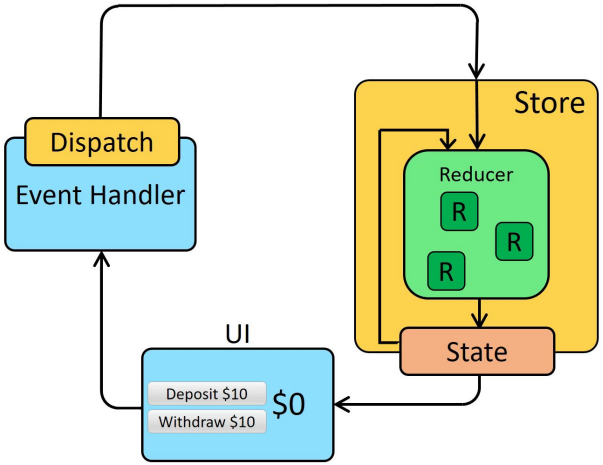
https://m3.material.io/

Redux Toolkit

Redux is a pattern and library for managing and updating **application state**, using events called "**actions**".

It serves as a centralized store for a **global state** (used across your entire application), with rules ensuring that the state can only be updated in a predictable fashion.

The patterns and tools provided by Redux make it easier to understand **when, where, why, and how the state in your application is being updated**, and how your application logic will behave when those changes occur.



https://redux-toolkit.js.org/

React Hook Form + Yup

```
import { useForm } from 'react-hook-form'

export const App = () => {
  const { register, handleSubmit, formState: { errors } } = useForm()
  const onSubmit = data => console.log(data)
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input type="text" placeholder="Name" {...register("Name", {required: true, maxLength: 80})} />
      <input type="email" placeholder="Email" {...register("Email", {required: true, pattern: /^\S+@\S+$/i})} />
      <input type="submit" />
    </form>
  )
}
```

https://react-hook-form.com/

https://github.com/jquense/yup

react-i18next

Internationalization framework for React which is based on i18next.

```
import i18n from "i18next"
import { useTranslation, initReactI18next } from "react-i18next"

i18n
  .use(initReactI18next) // passes i18n down to react-i18next
  .init({
    // [...]
  })

export const App = () => {
  const { t } = useTranslation()
  return <h2>{t('Welcome to React')}</h2>
}
```

https://react.i18next.com/

Other noteworthy packages

- **Axios** - https://axios-http.com/
  Promise based HTTP client for the browser and node.js
- **Lodash** - https://lodash.com/
  A modern JavaScript utility library delivering modularity, performance & extras
- **Luxon** - https://moment.github.io/luxon/
  Library for dealing with dates and times in JavaScript
- **React Draggable** - https://github.com/react-grid-layout/react-draggable
  A simple component for making elements draggable
- **React Motion** - https://github.com/chenglou/react-motion
  Popular animation library

Good coding!

You can find these slides and the code of the sample application on the following GitHub repository:

https://github.com/franksacco/react-in-practice

For doubts or further information, please contact:

- Francesco Saccani - francesco.saccani@unipr.it
- Gabriele Penzotti - gabriele.penzotti@unipr.it

→ **Projects & Thesis proposals**