



**Become a Java Full Stack Developer
with React & Spring Boot**

Why Full Stack ?



Let's talk about why Full Stack is awesome (apart from sounding cool at parties 😎).

- ✓ Full Stack Developers are like the Swiss Army knives of software. You can build the backend, frontend, and still have time to name your variables properly... (okay, maybe not that much time 😅).
- ✓ High demand, high salary, and low boredom – what more can you ask for?
- ✓ Whether you're building e-commerce apps, social networks, or the next food delivery unicorn 🐄 – Full Stack is where the magic happens.

What You'll Learn – Course Agenda



What You Really Need (Pre-requisites):



- ✓ Java, Javascript
- ✓ HTML, CSS
- ✓ Willingness to Learn and Break Things
- ✓ A Laptop (That Doesn't Cry When You Open IntelliJ or VS Code)

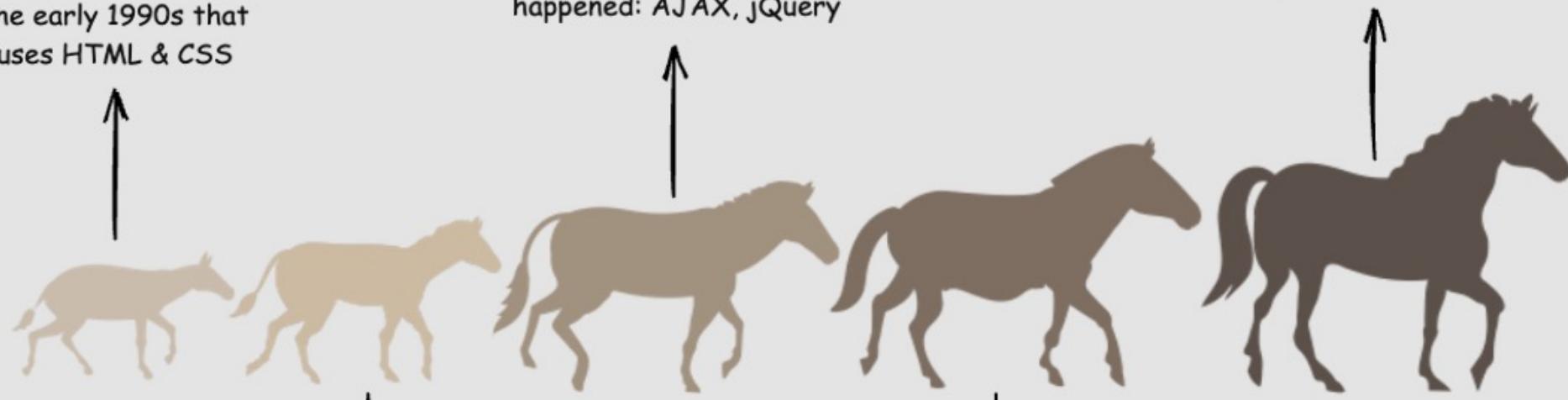
Evolution of Web Apps

How web applications have evolved over time !!!

Static websites of
the early 1990s that
uses HTML & CSS

By the mid-2000s,
something revolutionary
happened: AJAX, jQuery

Now we come to the star of this
course: Full Stack Applications.
Key technologies used are React,
SpringBoot, MySQL etc.



Dynamic websites came in
during early 2000s that
uses HTML, CSS, JS,
PHP, ASP.NET, JSP, and
databases like MySQL.

Fast forward to the
2010s, and we see the
emergence of Single-Page
Applications or SPAs.
Frameworks like Angular,
React, and Vue.js came
into lime light

The Fullstack Toolkit: Gearing Up for Awesomeness!

eazy
bytes

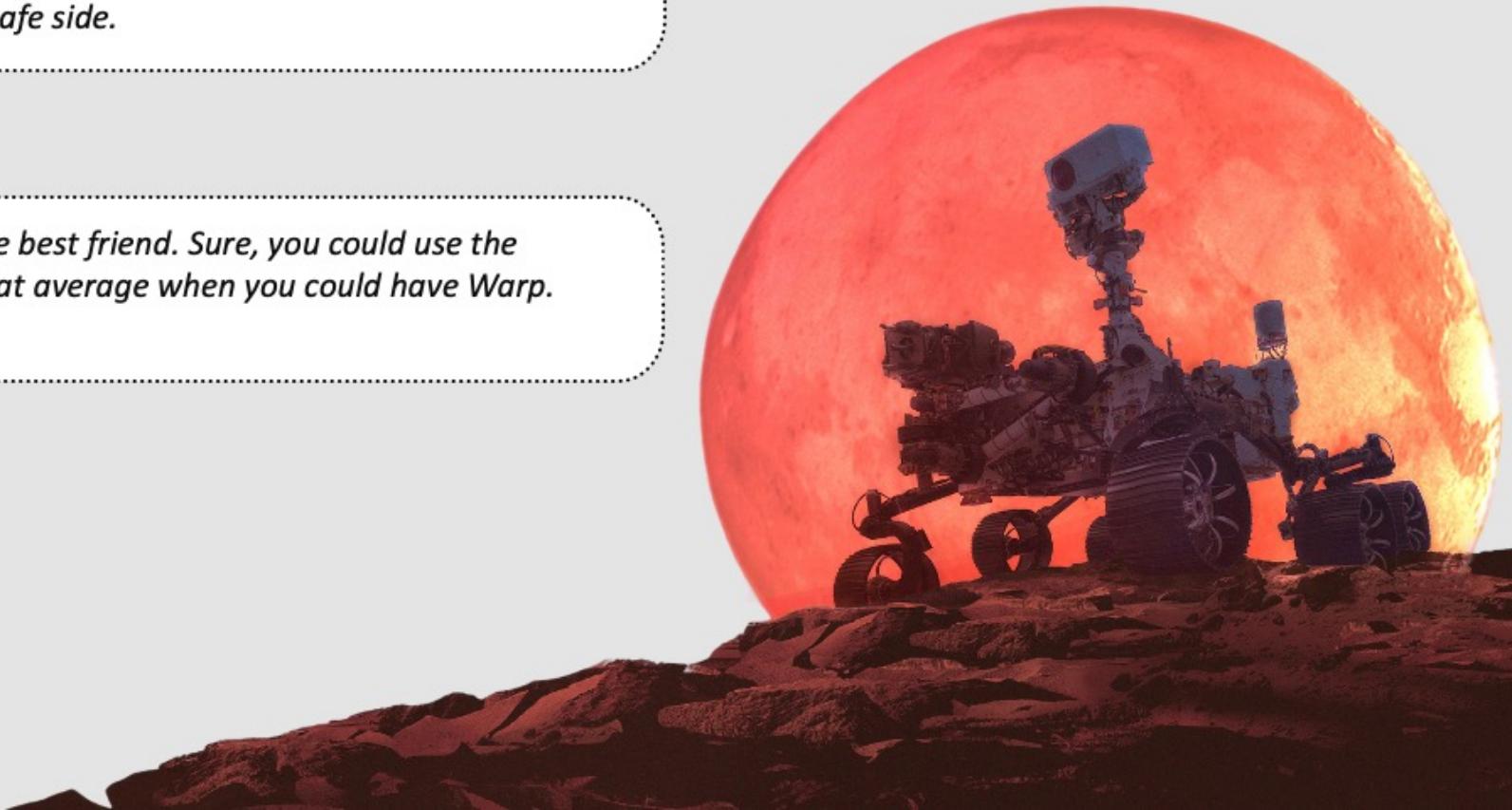
Hey students! Ready to dive into the magical world of Fullstack? Before we start, let's assemble our superhero toolkit. These are the essentials you need to code like a pro (and avoid pulling your hair out when something breaks).

Install Node.js

Command-line wizards like npm, npx help us create React apps, manage packages, and basically make React development as smooth as butter. To use them we need to download node from nodejs.org Pro tip: Go for the LTS version to stay on the safe side.

Warp/Any Terminal

Finally, let's talk about the terminal—your command-line best friend. Sure, you could use the default terminal that comes with your OS. But why stop at average when you could have Warp. Install Warp from warp.dev



The Fullstack Toolkit: Gearing Up for Awesomeness!

eazy
bytes

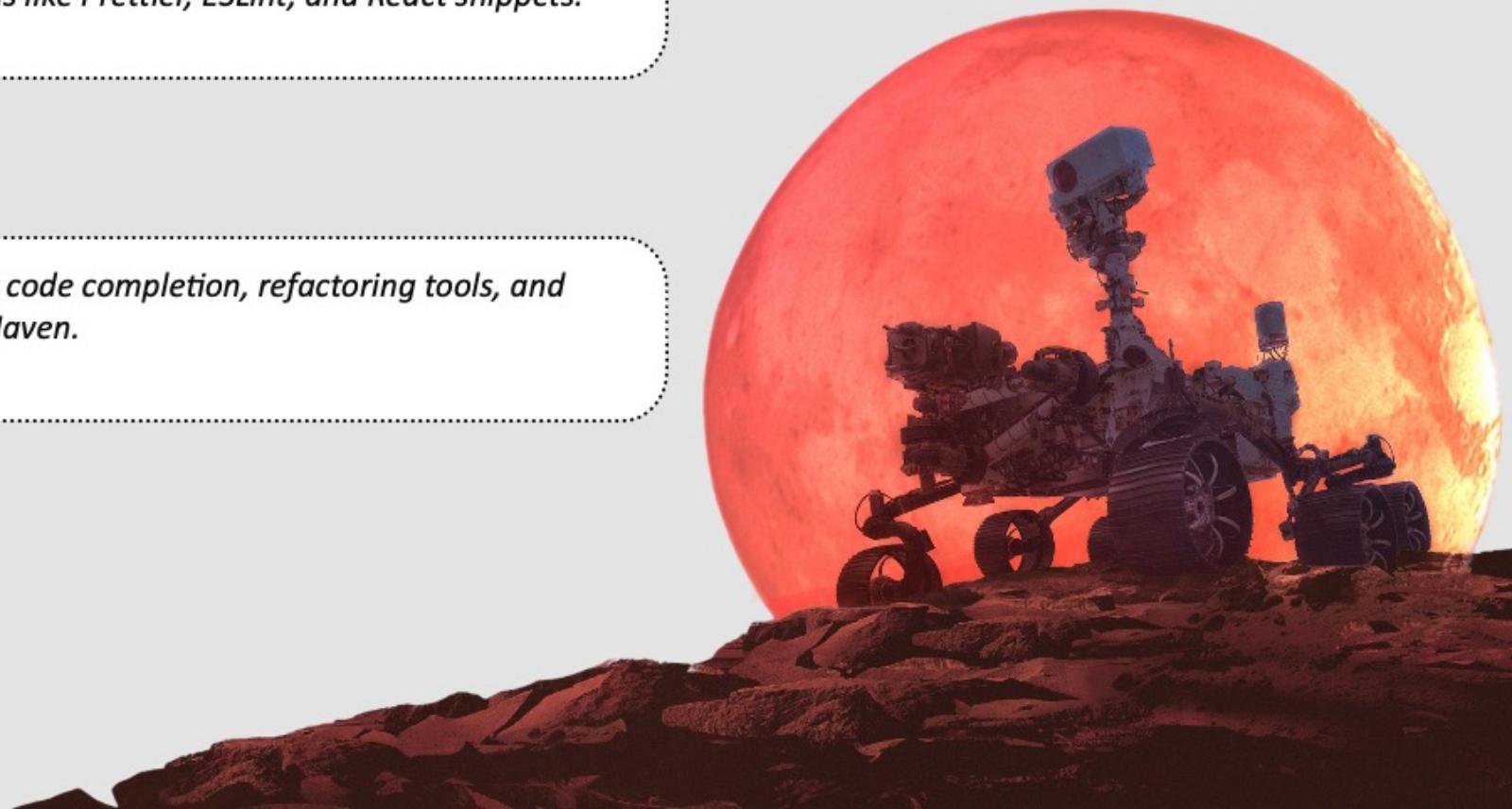
Hey students! Ready to dive into the magical world of Fullstack? Before we start, let's assemble our superhero toolkit. These are the essentials you need to code like a pro (and avoid pulling your hair out when something breaks).

Install VSCode

*Next, meet **Visual Studio Code**—aka the best code editor out there. Install VSCode from code.visualstudio.com and don't forget to add extensions like Prettier, ESLint, and React snippets. Trust me, your future self will thank you!*

Install IntelliJ IDEA

IntelliJ IDEA is a powerful Java IDE, known for intelligent code completion, refactoring tools, and seamless integration with frameworks like Spring and Maven. Install IntelliJ from [jetbrains.com/idea](https://www.jetbrains.com/idea/)



Most Useful VS Code Shortcuts for Developers (Mac & Windows)

eazy
bytes

General Shortcuts

Action	Mac	Windows/Linux
Command Palette	Cmd + Shift + P	Ctrl + Shift + P
Open Settings	Cmd + ,	Ctrl + ,
Open File	Cmd + P	Ctrl + P
Toggle Sidebar	Cmd + B	Ctrl + B
Show All Shortcuts	Cmd + K Cmd + S	Ctrl + K Ctrl + S
Toggle Terminal	Ctrl + ~	Ctrl + ~
Toggle Fullscreen	Ctrl + Cmd + F	F11
Close Current Tab	Cmd + W	Ctrl + W

Most Useful VS Code Shortcuts for Developers (Mac & Windows)

eazy
bytes

Editing Shortcuts

Action	Mac	Windows/Linux
Copy Line Down	Shift + Option + ↓	Shift + Alt + ↓
Copy Line Up	Shift + Option + ↑	Shift + Alt + ↑
Delete Line	Cmd + Shift + K	Ctrl + Shift + K
Move Line Down	Option + ↓	Alt + ↓
Move Line Up	Option + ↑	Alt + ↑
Comment/Uncomment	Cmd + /	Ctrl + /
Format Document	Shift + Option + F	Shift + Alt + F
Go to Line	Ctrl + G	Ctrl + G

Most Useful VS Code Shortcuts for Developers (Mac & Windows)

eazy
bytes

Navigation Shortcuts

Action	Mac	Windows/Linux
Navigate Between Tabs	Ctrl + Tab	Ctrl + Tab
Go to Definition	F12	F12
Peek Definition	Option + F12	Alt + F12
Find in File	Cmd + F	Ctrl + F
Find in All Files	Cmd + Shift + F	Ctrl + Shift + F
Quick Switch Editor	Cmd + 1/2/3 C	Ctrl + 1/2/3

Debugging Shortcuts

Action	Mac	Windows/Linux
Start/Stop Debugging	F5	F5
Step Over	F10	F10
Step Into	F11	F11
Continue	Shift + F5	Shift + F5
Toggle Breakpoint	F9	F9

Most Useful VS Code Shortcuts for Developers (Mac & Windows)

eazy
bytes

Search and Replace

Action	Mac	Windows/Linux
Find	Cmd + F	Ctrl + F
Replace	Cmd + Option + F	Ctrl + H
Find Next	Cmd + G	F3
Find Previous	Cmd + Shift + G	Shift + F3

File and Window Management

Action	Mac	Windows/Linux
New File	Cmd + N	Ctrl + N
Open File	Cmd + O	Ctrl + O
Save File	Cmd + S	Ctrl + S
Save All Files	Cmd + Option + S	Ctrl + Shift + S
Close Window	Cmd + Shift + W	Ctrl + Shift + W

Most Useful VS Code Shortcuts for Developers (Mac & Windows)

eazy
bytes

Terminal Shortcuts

Action	Mac	Windows/Linux
Toggle Terminal	Ctrl + ~	Ctrl + ~
New Terminal	Ctrl + Shift + ~	Ctrl + Shift + ~
Kill Terminal	Ctrl + C	Ctrl + C

Multi-Cursor Shortcuts

Action	Mac	Windows/Linux
Add Cursor Above	Cmd + Option + ↑	Ctrl + Alt + ↑
Add Cursor Below	Cmd + Option + ↓	Ctrl + Alt + ↓
Select Next Match	Cmd + D	Ctrl + D
Select All Matches	Cmd + Shift + L	Ctrl + Shift + L

💡 Pro Tip:

- You can customize your shortcuts in Preferences → Keyboard Shortcuts (Cmd/Ctrl + K Cmd/Ctrl + S).
- Use Command Palette (Cmd/Ctrl + Shift + P) to search and execute commands quickly.

When working in programming, you may have likely come across the terms "**Library**" and "**Framework**"—like a **Javascript library** or a **Java framework**. But have you ever stopped to wonder:

- ✓ What's the difference between a library and a framework?
- ✓ Are they interchangeable?
- ✓ Which one should you choose for your project?

Let's clear up the confusion once and for all!

Both libraries and frameworks are pre-written code components created by experienced developers to solve common problems efficiently. Their goal is the same: to simplify development and reduce repetitive tasks.

Simple Explanation

Library: A library is a collection of reusable code, functions, or modules that you can call when needed. You are in control of the flow of the program and decide when and how to use the library.

Framework: A framework provides a complete structure and controls the flow of the program. It decides when and how your code is executed. You follow the framework's rules and structure.

- ✓ **Library** a collection of reusable code designed to perform specific tasks (e.g., manipulating strings, handling HTTP requests, or working with dates).
- ✓ You call the library's methods/functions directly from your code.
- ✓ It offers flexibility—you choose which parts to use and when.
- ✓ Typically easier to learn because you only need to understand the functions you plan to use.
- ✓ Best suited for modular tasks where control remains with the developer.
- ✓ Examples include React (JS), Lombok (Java), TensorFlow (Python)



- ✓ **Framework** provides a foundation and architecture for building software applications.
- ✓ It follows the principle of Inversion of Control (IoC)—the framework calls your code, not the other way around.
- ✓ Frameworks impose structure—you follow predefined rules and flow
- ✓ Often has a steeper learning curve because it requires understanding the entire architecture.
- ✓ Best suited for large-scale, complex applications where consistency and structure are critical.
- ✓ Examples include Angular (JS), SpringBoot(Java), Django (Python)

🎓 Quick Takeaway for Students

- If you're calling the code → **Library**
- If the code is calling you → **Framework**

🛠️ Analogy: Building a House 🏠

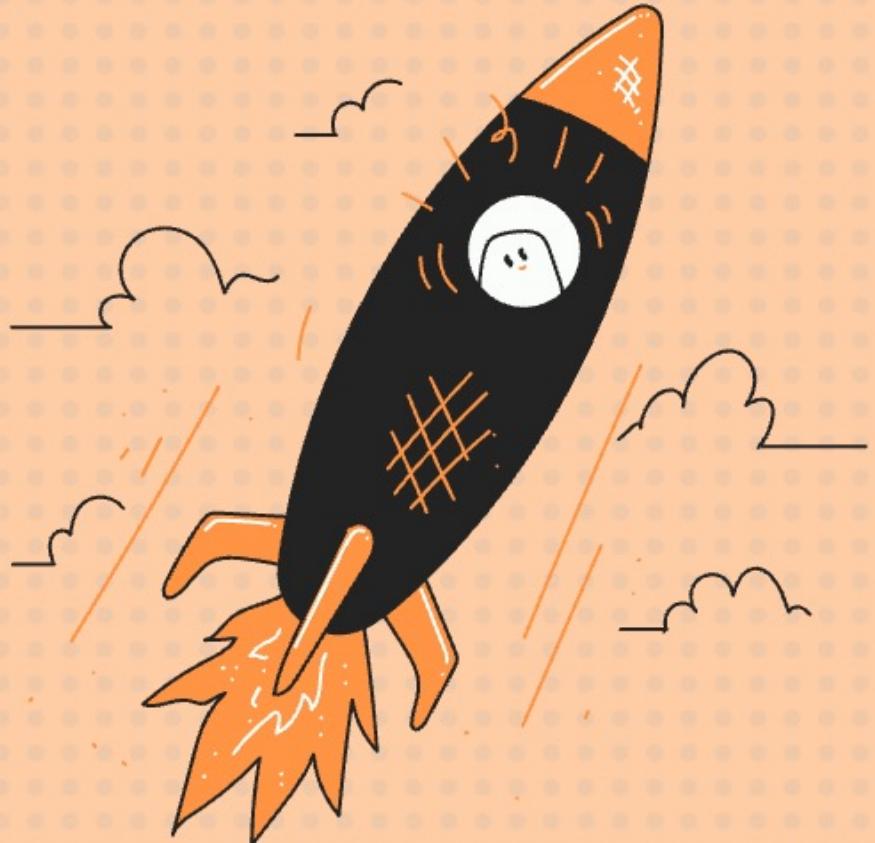
Imagine you're building a house:

Library: It's like having a toolbox 📦 with various tools (hammer, screwdriver, nails). You pick the tools you need, decide how to use them, and you're in control of building the house.

Framework: It's like having a pre-built construction scaffold 🏒 and a step-by-step guide. The framework sets the structure, and you can only build your house within the scaffold. You follow its instructions, and it tells you when and where to add your contributions.

What is React

React is a JavaScript library used for building user interfaces (UIs), especially single-page applications (SPAs).



What is React

- ✓ Developed by **Facebook** (now Meta)
- ✓ Focuses on **building reusable UI components**
- ✓ Efficiently updates and renders UI with a **Virtual DOM**
- ✓ Follows a **component-based architecture**



Key Features of React

- ✓ **Component-Based:** Build UIs using small, reusable pieces called **components**.
- ✓ **Declarative Syntax:** Describe **what you want**, React figures out **how to do it**.
- ✓ **Virtual DOM:** Faster updates without directly manipulating the browser DOM.
- ✓ **Unidirectional Data Flow:** Data flows in **one direction**, making it easier to debug.
- ✓ **JSX (JavaScript XML):** Write HTML-like syntax in JavaScript.

Why use React

- ✓ **Reusable Components:** Write once, use multiple times.
- ✓ **Fast Rendering:** Thanks to the **Virtual DOM**.
- ✓ **Scalable Applications:** Easy to manage even in large projects.
- ✓ **Strong Community Support:** Backed by Facebook and a global developer community.



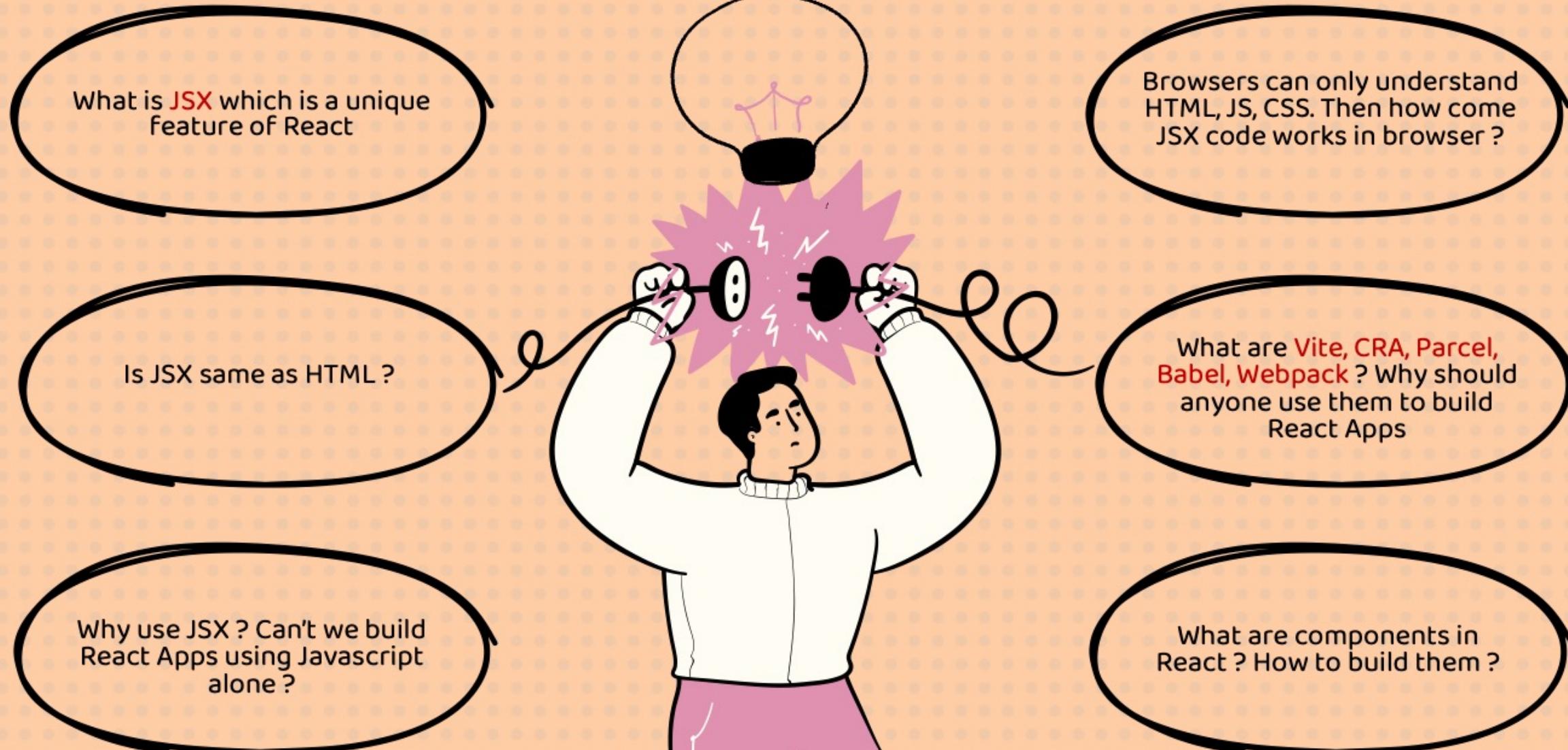
How React works

- ✓ Write Components
- ✓ React Creates Virtual DOM
- ✓ Compare with Real DOM
- ✓ Update Only Changed Parts

This makes React **fast and efficient!** ⚡



Common questions while learning React ?





Building Apps using plain HTML & CSS

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Simple HTML</title>
    <link rel="stylesheet" href=".//index.css">
</head>
<body>
    <div id="root">
        <h1>Hello World !!!</h1>
    </div>
</body>
</html>
```

Complete static web app.
There will not be any dynamic behaviour



Building Apps using plain HTML, CSS & JS

```
<body>

    <div id="root"></div>
    <script>
        const randomNumber = Math.floor(Math.random() * 100) + 1;
        const numberElement = document.createElement("h1");
        numberElement.innerHTML = `Generated Number:
        ${randomNumber}`;

        const resultElement = document.createElement("h2");
        if (randomNumber % 2 === 0) {
            resultElement.innerHTML = "The number is even!";
        } else {
            resultElement.innerHTML = "The number is odd!";
        }

        const root = document.getElementById("root");
        root.appendChild(numberElement);
        root.appendChild(resultElement);
    </script>

</body>
```

Javascript makes apps to behave dynamically and allows DOM manipulation



Building Apps using plain HTML, CSS, JS & React

```
<body>
    <script src="./react-lib/react.development.js"></script>
    <script src="./react-lib/react-dom.development.js"></script>
<div id="root"></div>
<script>
    const randomNumber = Math.floor(Math.random() * 100) + 1;
    const heading = React.createElement("h1",null,
        `Generated Number: ${randomNumber}`);

    const result = React.createElement("h2",null, randomNumber %
        2 === 0 ? "The number is even!" : "The number is odd!");

    const root = ReactDOM.createRoot(document.getElementById("app"));
    root.render(React.createElement("div", null,
        [heading, result]));

</script>

</body>
```

Using React provides a more efficient, modular, and maintainable way to build interactive UIs with a virtual DOM, component-based architecture, and reusable code.

In React, `createElement` and `createRoot` are methods that allow you to interact with the DOM and create elements dynamically.

React.createElement()

This method is used to create React elements (JSX-like elements) in JavaScript. It is a core function that React uses under the hood to create elements that are later rendered to the DOM.

syntax: `React.createElement(type, [props], [...children])`

type: A string representing the type of HTML element (e.g., "div", "h1", "p", etc.) or a React component.

props: An object that contains the attributes or properties to be applied to the element (e.g., { id: "root", className: "my-class" }).

children: The children inside the element (other React elements or plain text).

Example: `React.createElement('h1', null, 'Hello World');`

This creates an `<h1>` element with the content "Hello World". In JSX syntax, this would look like:

`<h1>Hello World</h1>`



ReactDOM.createRoot()

This method initializes the React rendering system by creating a React root and attaching it to a specific DOM node. It serves as the entry point for rendering React components or element trees into the DOM. The process involves setting up a root container for the application (or a part of it) and rendering React components within this container efficiently.

Syntax:

```
const root = ReactDOM.createRoot(container);
root.render(element);
```

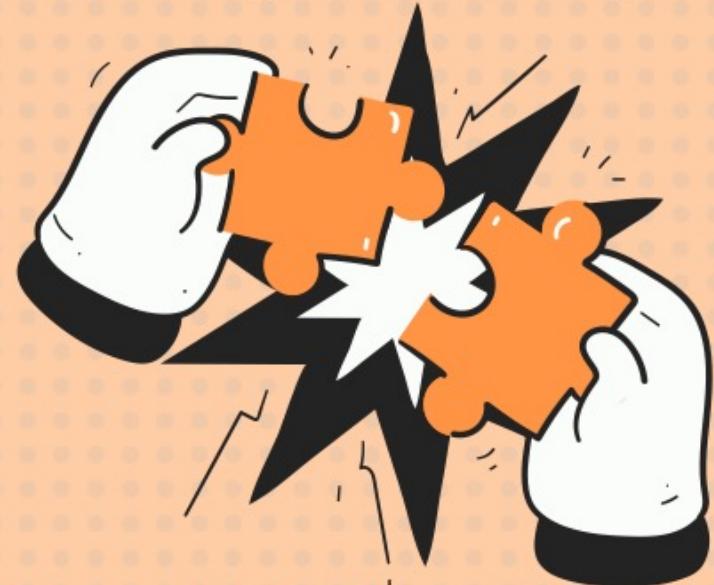
container: The DOM element where you want to attach the React application (e.g., `document.getElementById('root')`).

element: The React element that will be rendered inside the container.

Example:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement('h1', null, 'Hello React!'));
```

In a typical React application, `createElement` is used to define what the UI looks like, while `createRoot` is used to attach the UI to the actual web page.



What is JSX

- ✓ **JSX (JavaScript XML)** is a syntax extension for JavaScript.
- ✓ It allows you to write HTML-like code directly inside JavaScript.
- ✓ Used primarily in React to define UI components.

1) Basic JSX Structure

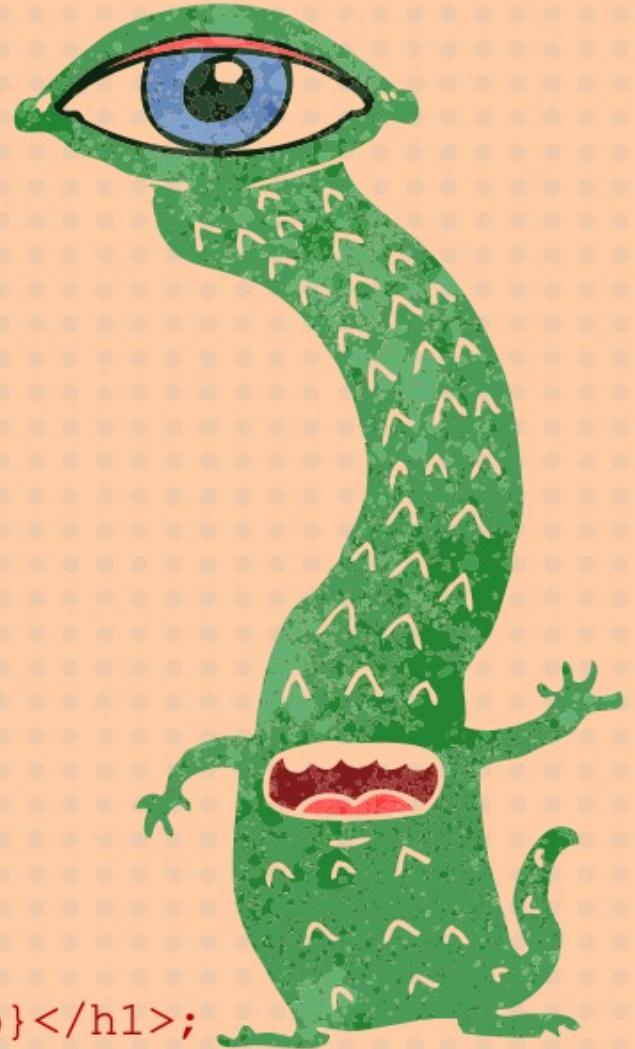
```
const element = <h1>Hello, World!</h1>;
```

2) JSX with Variables

```
const name = "John Doe";
const element = <h1>Hello, {name}!</h1>;
```

3) JSX with Expressions

```
const a = 5;
const b = 3;
const element = <h1>The sum of {a} and {b} is {a + b}</h1>;
```



4) JSX with Conditional Rendering

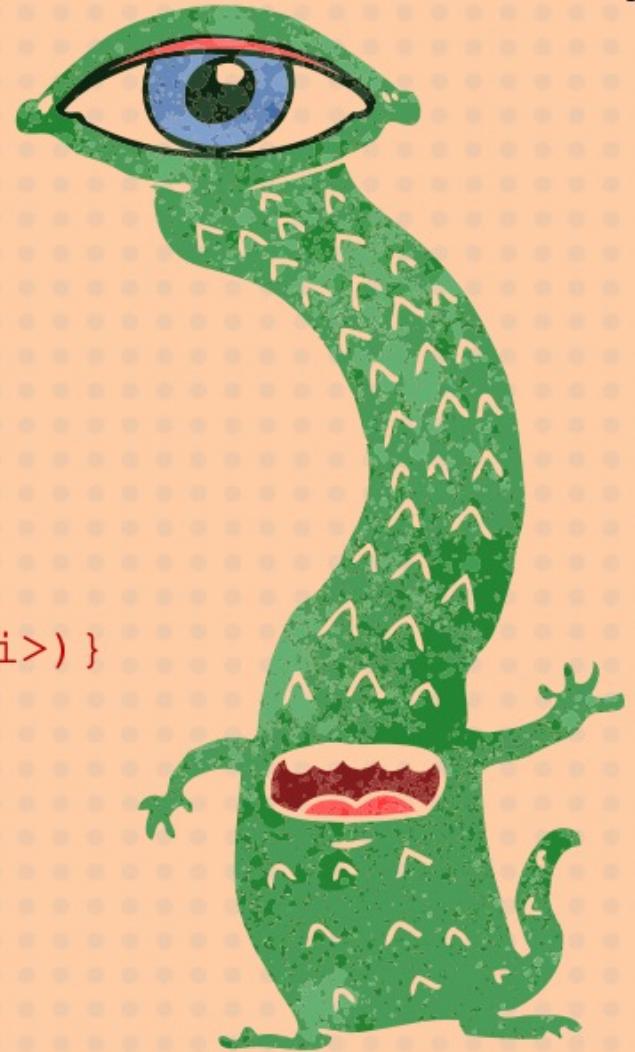
```
const isLoggedIn = true;
const element = (
  <h1>{isLoggedIn ? "Welcome back!" :
    "Please sign up."}</h1>
);
```

5) JSX with Lists (Rendering Arrays)

```
const items = ['Apple', 'Banana', 'Cherry'];
const element = (
  <ul>
    {items.map(item => <li key={item}>{item}</li>)}
  </ul>
);
```

6) JSX with Function Components

```
function Greeting() {
  return <h1>Hello, welcome to React!</h1>;
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Greeting />);
```



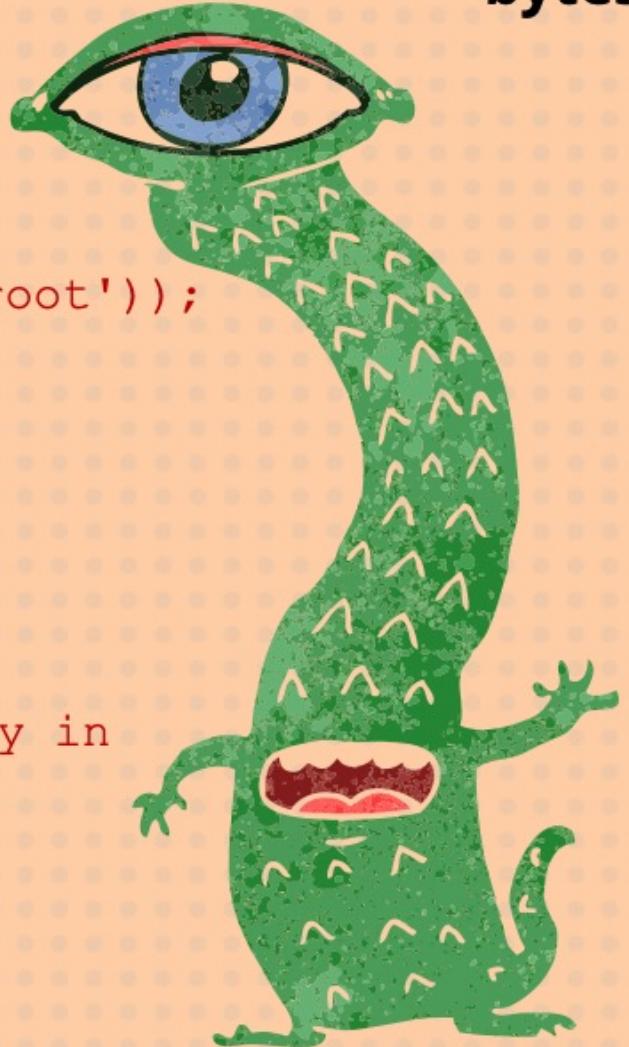
7) JSX with Props

```
function Welcome(props) {  
  return <h1>Hello, {props.name}!</h1>;  
}
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Welcome name="Alice" />);
```

8) JSX with Nested Elements

```
const element = (  
  <div>  
    <h1>Welcome to JSX</h1>  
    <p>JSX allows you to write HTML elements directly in  
       JavaScript!</p>  
  </div>  
) ;
```

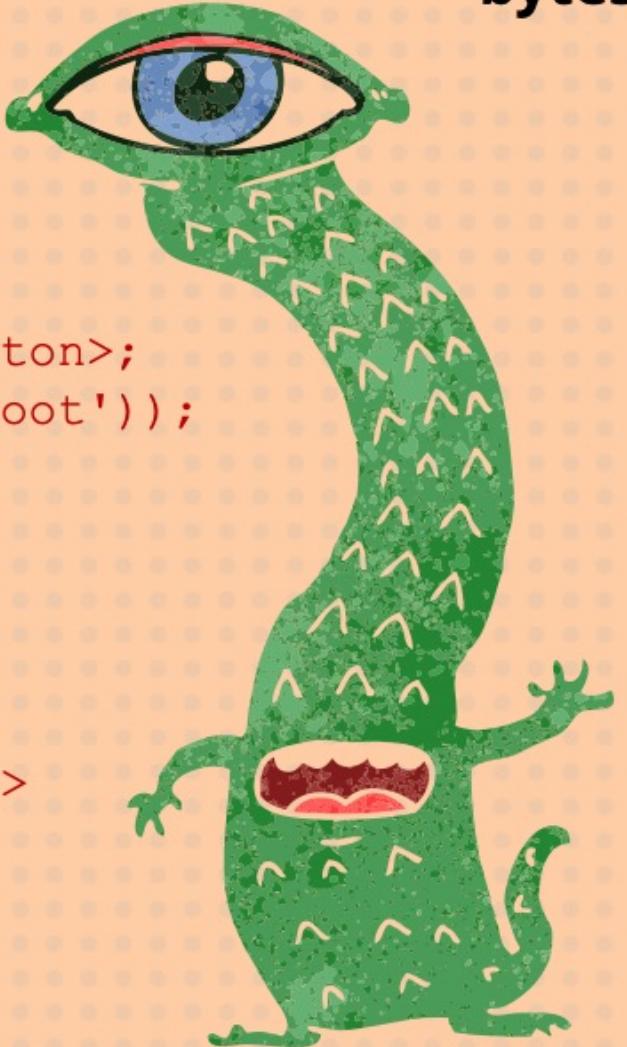


9) JSX with Event Handlers

```
function handleClick() {  
  alert('Button clicked!');  
}  
  
const element = <button onClick={handleClick}>Click me</button>;  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(element);
```

10) JSX with Styling

```
const element = (  
  <div style={{ color: 'blue', backgroundColor: 'yellow' }}>  
    This is styled using inline styles in JSX!  
  </div>  
);
```



Why JSX

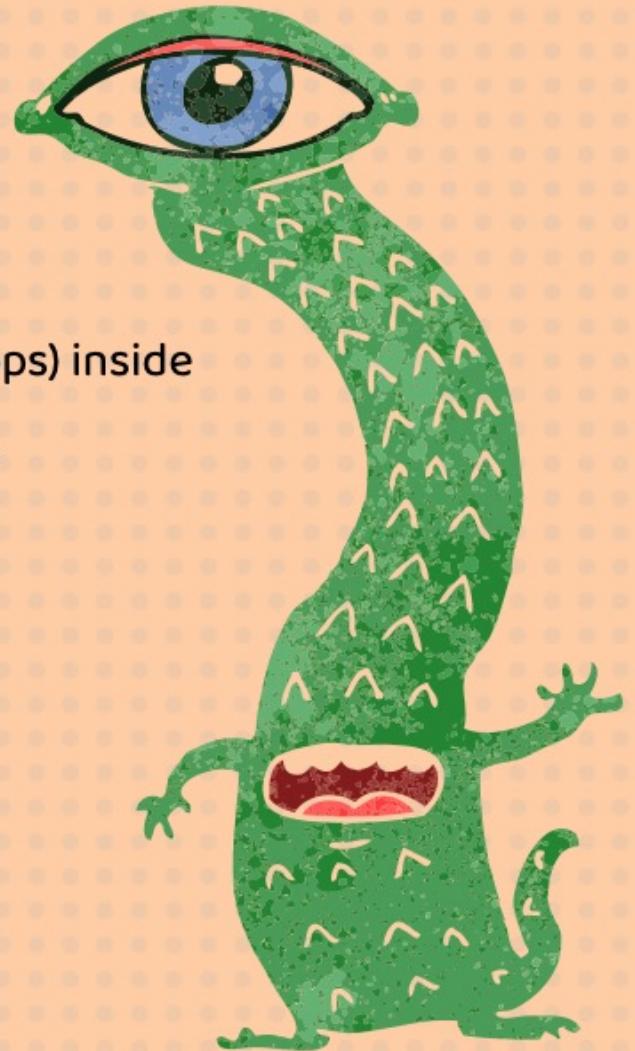
- ✓ Readable Code: Easier to understand UI structure.
- ✓ Combines HTML & JavaScript Logic: Write both in one place.
- ✓ Prevents Injection Attacks: JSX is safe by default.
- ✓ Allows to Embed expressions and variables, use JavaScript logic (conditions, loops) inside JSX, define and use components, pass data with props, and bind event handlers.
- ✓ More **efficient rendering** with Virtual DOM

JSX Rules to Remember

- ✓ Should return a Single Parent Element:
- ✓ Use Curly Braces {} For JavaScript expressions.
- ✓ Self-Closing Tags are Mandatory

✓ Correct in JSX `const example = ;`

✗ Incorrect in JSX `const example3 = ;`





How JSX is Converted into JavaScript

Browsers can't understand JSX directly. JSX needs to be transformed into regular JavaScript that the browser can execute.

Transpilation process takes care of converting JSX to JS

- ✓ Transpilation stands for "source-to-source compilation". It is the process of taking source code written in one programming language or syntax (like JSX or modern JavaScript) and converting it into another form of source code (like older JavaScript that browsers can understand).

Examples include JSX is converted into JavaScript, Modern JavaScript features (like `async/await`, arrow functions, etc.) might be transpiled into older JavaScript syntax for compatibility with older browsers.

Compilation: While "compiling" often refers to converting code into machine code or bytecode (used by lower-level languages like C or Java), transpiling is a form of compilation specifically designed to convert one high-level language (or syntax) into another.



Who handles the transpilation ?

Tools like Babel, ESBuild, Webpack, Rollup, Vite etc. takes care of transpilation during the build process.

Vite uses esbuild for JavaScript and TypeScript transpilation. esbuild is significantly faster than Babel because it's written in Go and optimized for speed.

Why Use Vite if ESBuild already handles Transpilation?

While ESBuild excels at transpiling JavaScript and TypeScript quickly, Vite is not just about transpilation. It provides a complete modern frontend development experience that goes beyond just ESBuild's capabilities.

ESBuild is Only a Tool; Vite is a Build Toolchain.

ESBuild is like a high-speed engine. Vite is a full car with a powerful engine, navigation system, and comfy seats.

Vite: A Frontend build tool and development server built *on top of ESBuild* for transpilation, but it also handles:

- Dependency pre-bundling
- Hot Module Replacement (HMR)
- Optimized production builds
- Static asset handling (CSS, images, etc.)

What is Vite?

Vite (pronounced 'veet'—like wheat without the 'wh') is a lightning-fast frontend build tool. Think of it as the Ferrari of app generators, but without the price tag.

Why should I use a build tool?

While it's technically possible to set up React without a build tool, it's not practical for modern development. Here's why:

- 1. JSX Support:** JSX isn't valid JavaScript, and browsers can't interpret it directly. You'd need tools like esBuild, Babel to transpile it into browser-friendly code.
- 2. Dependency Management:** React apps rely on npm packages (React, React DOM, etc.), requiring manual setup, version management, and updates without a build tool.
- 3. Developer Experience:**
 - No Hot Module Replacement (HMR), so you'd refresh the browser manually for every change.
 - No fast builds, leading to slower reloads as the app grows.
 - No error overlays to debug issues efficiently.

Tools like Vite handle all of this seamlessly, saving time and improving performance.

Vite: The Flash of React App Generators

Steps to Generate a React App using Vite

```
> npm create vite@latest eazystore-ui
  Select a framework: > React
  Select a variant: > JavaScript
> cd my-app
> npm install
> npm run dev
```

The terminal will display a local development server URL (e.g., <http://localhost:5173>). Open it in your browser to view your app.

Folder and File Structure of a React App Generated by Vite

eazy
bytes

node_modules

*contains all the dependencies installed via npm.
Automatically managed, so you don't need to touch
this folder.*

public

*Stores static assets like images and icons. These assets
are not processed by Vite, so they must be referenced
directly using relative paths.*

src

This is where your application's source code lives.

assets

*A folder to store images, fonts, or other assets that you
want to import dynamically into your components.*

index.html

*The main HTML file for the app. Vite injects your React
app's JavaScript code into this file during development
and production builds.*

MY-APP

- > node_modules
- > public
- < src
 - > assets
 - # App.css
 - ✳ App.jsx
 - # index.css
 - ✳ main.jsx
 - ❖ .gitignore
 - eslint.config.js
 - <> index.html
 - { } package-lock.json
 - { } package.json
 - ⓘ README.md
 - JS vite.config.js

package.json

*Manages the dependencies, scripts, and metadata for
your project.*

vite.config.js

*Vite's configuration file where you can customize the
build process, aliases, plugins, and more*

.gitignore, README.md

*Specifies files and directories that Git should ignore when
tracking Provides documentation and instructions about
the project*

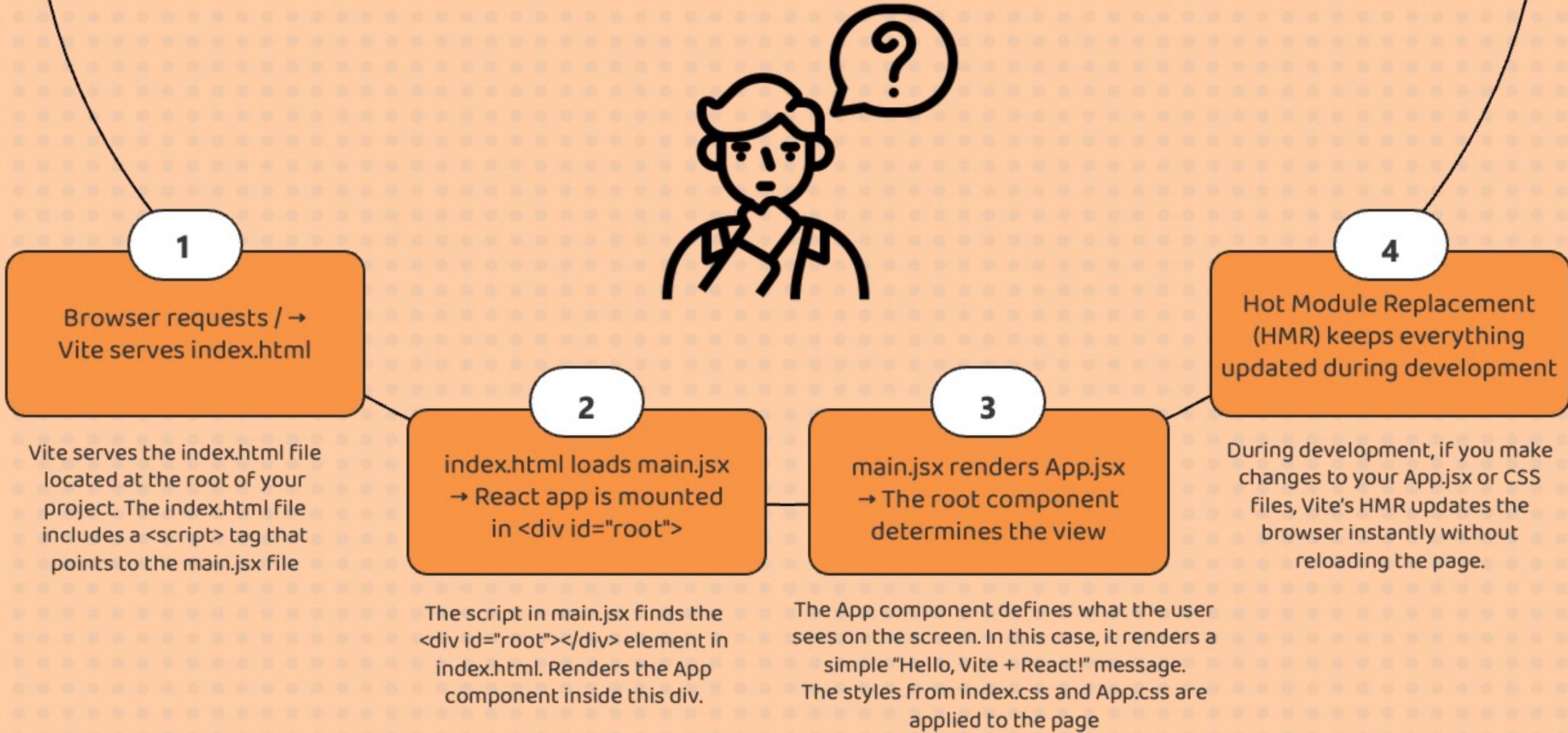
eslint.config.js

*Configure ESLint (a popular JavaScript/TypeScript linting
tool) for your project. This file allows you to define
specific rules, plugins, and settings for ESLint*

package-lock.json

*It ensures that your project's dependencies are consistent
and reproducible across different environments by locking
specific versions of installed packages.*

When a user accesses the root path (/), here's what happens step-by-step:

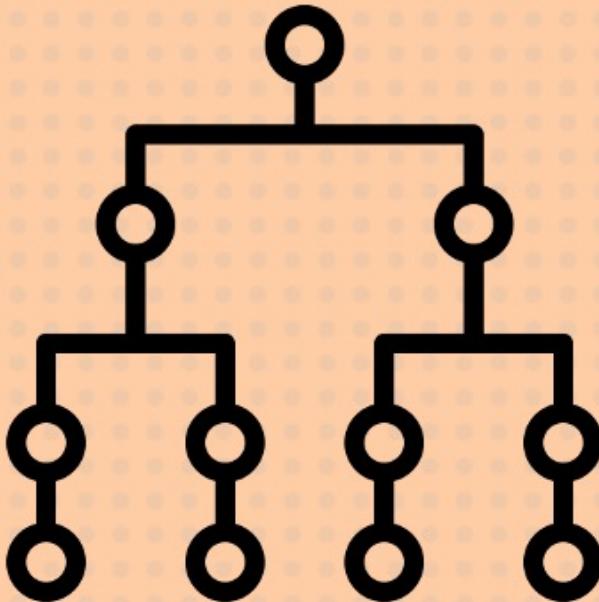


What is DOM

- ✓ DOM (Document Object Model) represents the structure of a web page as a tree of objects. It allows JavaScript to interact with and manipulate HTML elements dynamically.

Sample HTML

```
<div>
  <h1>Hello World</h1>
  <p>This is a paragraph</p>
</div>
```



Visual Representation

Document

```
  └── <div>
      ├── <h1>Hello World</h1>
      └── <p>This is a paragraph</p>
```

Key Points:

- ✓ Standard way to represent HTML.
- ✓ Allows real-time updates via JavaScript.
- ✓ Problem: Updating the DOM directly is slow and expensive.

What is Virtual DOM

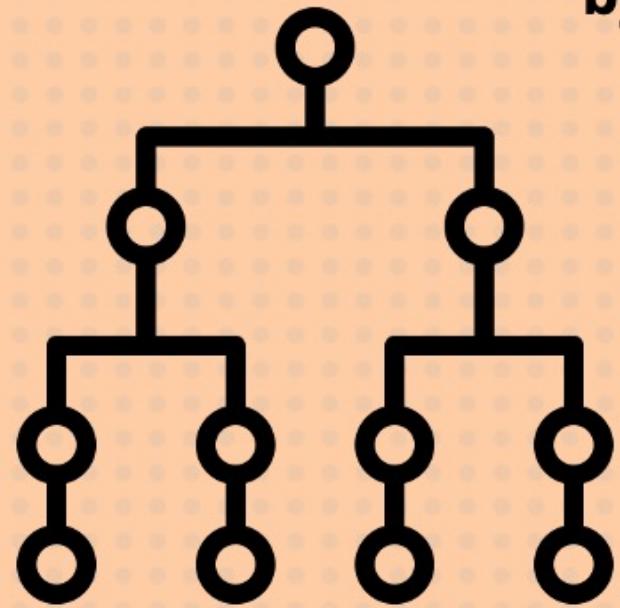
- ✓ Virtual DOM is a lightweight, in-memory representation of the real DOM. Think of it as a blueprint or snapshot of the actual DOM. React uses Virtual DOM to make updates more efficient.

How it works

- React updates the Virtual DOM when state changes.
- React compares (diffs) the updated Virtual DOM with the previous version.
- It identifies the minimal changes needed.
- Only the necessary parts of the real DOM are updated.
- React State → Virtual DOM → Diff → Real DOM Update

Why React uses Virtual DOM ?

- Faster Updates: Only changes are applied to the real DOM.
- Efficient Rendering: Avoid unnecessary re-renders.
- Better Performance: React manages updates smartly.
- Cleaner Code: Developers focus on what to update, not how to update.



- ✓ React uses **reconciliation algorithm** to update the DOM efficiently. It determines the minimum number of changes needed to update the Real DOM.
- ✓ React **Rendering** is the process of converting React components into DOM elements.

DOM vs Virtual DOM

Aspect	DOM	Virtual DOM
Definition	Represents UI structure	Lightweight copy of DOM
Updates	Slow, updates entire tree	Fast, updates specific nodes
Efficiency	Direct manipulations	Uses diffing algorithm
Re-Renders	Frequent and costly	Batch and optimized
Usage	Native to browsers	Used by React

Semantic Versioning (SemVer) Basics

The symbols like ^, ~, and others in package.json dependencies are called SemVer (Semantic Versioning) Range Specifiers. They define the allowed range of versions for each dependency.

Semantic Versioning follows the format:

MAJOR . MINOR . PATCH

MAJOR: Introduces breaking changes.

MINOR: Adds new features without breaking existing functionality.

PATCH: Fixes bugs without changing functionality.

1.2.9 → **MAJOR: 1, MINOR: 2, PATCH: 9**



Version Specifiers in a Nutshell

^ (caret) → Update MINOR and PATCH safely. (Recommended for most libraries)

~ (tilde) → Update only PATCH.

No Symbol → Lock to an exact version.

Meaning of Symbols

Symbol	Example	Meaning
^	^1.2.9	Allows updates that do not change the MAJOR version. Example: 1.x.x but not 2.x.x.
~	~1.2.9	Allows updates that do not change the MINOR version. Example: 1.2.x but not 1.3.0.
>	>1.2.9	Allows versions greater than 1.2.9
>=	>=1.2.9	Allows versions greater than or equal to 1.2.9
<	<1.2.9	Allows versions less than 1.2.9
<=	<=1.2.9	Allows versions less than or equal to 1.2.9
*	*	Matches any version . (Rarely used in production)
x	1.x.x	Matches any version for the x position (wildcard).
latest	latest	Always installs the latest published version .

Best Practices

1. Use ^ for **libraries** and frameworks (**default behavior**) to stay updated with compatible MINOR and PATCH releases.
2. Use ~ for **tools or plugins** where patch-level updates are safe.
3. Use **exact versions** (1.2.9) for production-critical dependencies when stability is a must.

What is package-lock.json ?

It is a file automatically generated by npm when you run `npm install`. It locks the exact versions of dependencies installed in your project.

Why we need this file ?

In package.json, dependencies and peer dependencies typically use version ranges, not fixed versions. This makes `npm install` non-deterministic—running it today and again months later, or on different machines, may produce varying node_modules trees.

When multiple developers work on the same project, these inconsistencies can lead to dependency conflicts or breaking changes. This is why package-lock.json is crucial—it locks exact versions, ensuring a consistent and reliable dependency tree across environments.

Should we commit package-lock.json into GitHub ?

Absolutely! Committing package-lock.json ensures that every developer who clones your repository installs exactly the same dependency versions as in your environment. This helps replicate the Node.js setup consistently across different machines, preventing version mismatches, unexpected bugs, or breaking changes.



What are React Components

- ✓ React Components are the building blocks of a React application.
- ✓ They are reusable pieces of UI (User Interface).
- ✓ Each component can have its own logic and styling.
- ✓ Think of components like LEGO blocks that you can assemble to create a complete application.
- ✓ A React application is built like a tree of components, with the App component as the root that brings all the other components together.

Why use Components

- ✓ **Reusability:** Write once, use anywhere.
- ✓ **Separation of Concerns:** Each component handles a specific part of the UI.
- ✓ **Easier to Debug:** Smaller, independent pieces of code.
- ✓ **Better Collaboration:** Teams can work on different components simultaneously.

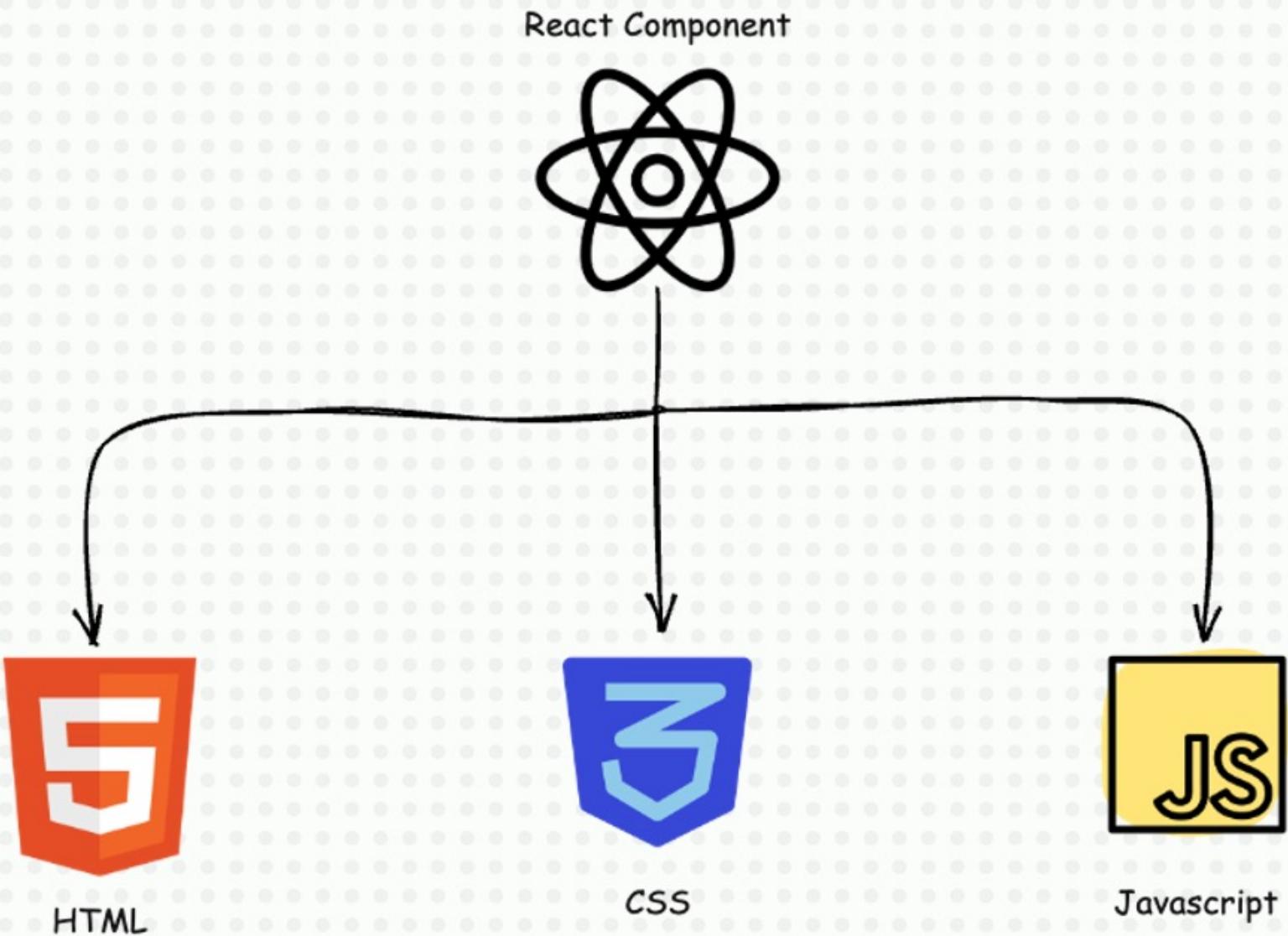


React Component Rules

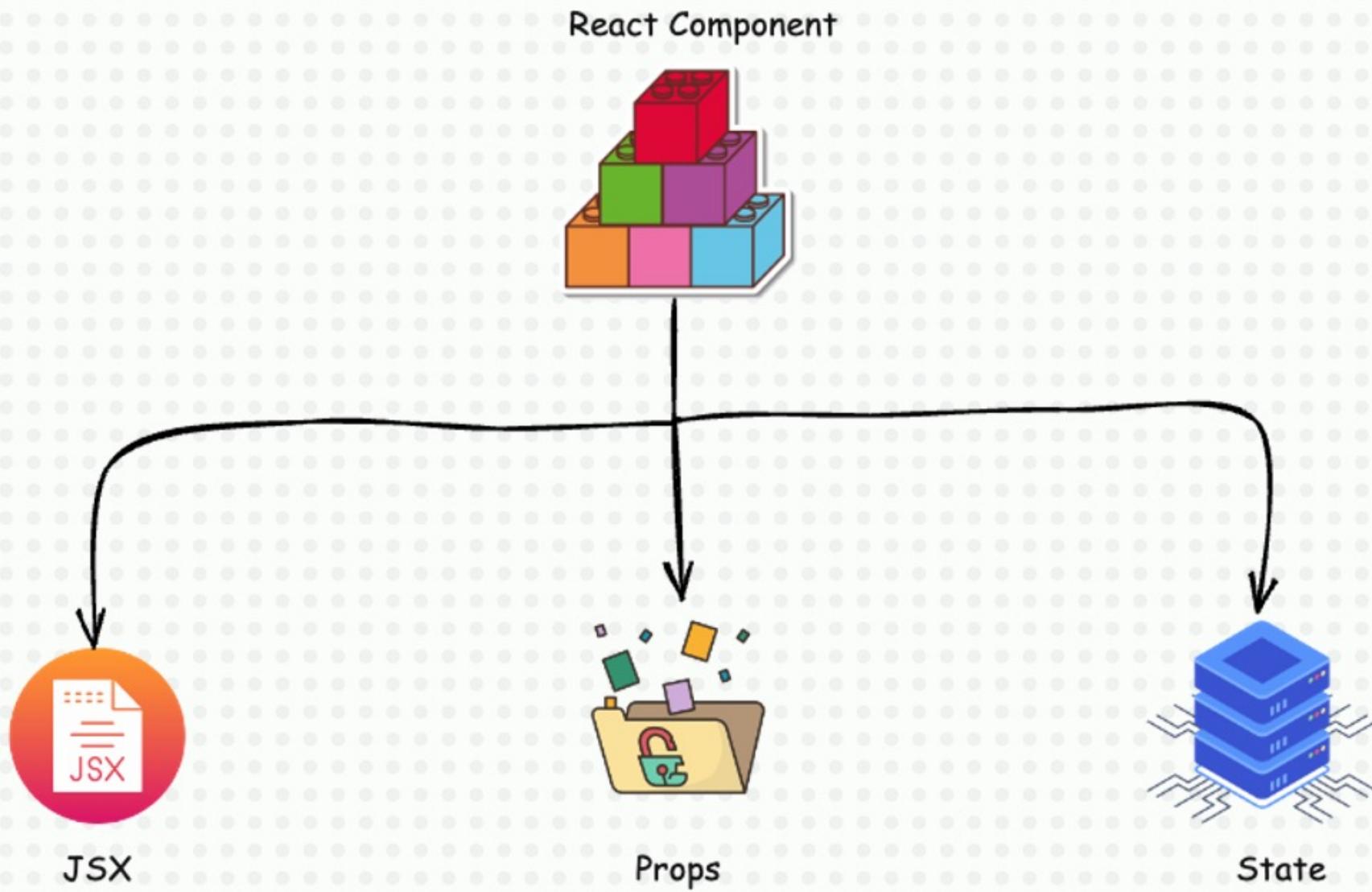
- ✓ React Component names should start with an uppercase letter.
- ✓ Choose a name that clearly describes the UI component (e.g., "Header" or "MyHeader").
- ✓ For multi-word names, use PascalCase formatting (e.g., "MyHeader").
- ✓ The component function must return a value that React can render (i.e., display on the screen).
- ✓ Typically, it returns JSX.
- ✓ It can also return a string, number, boolean, null, or an array of these values.



React Components = **HTML + CSS + JS(x)**



React Components = JSX + Props + State



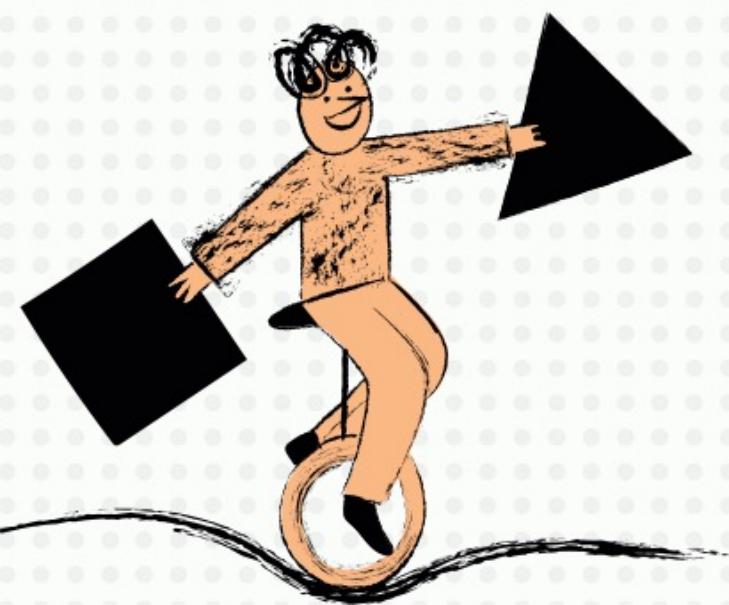


Which file extension should we use ?

While building React components, you can use either .js or .jsx extensions, but here's when to use each:

Use .jsx:

- ✓ If your file contains JSX (JavaScript XML), which is the syntax React uses to write HTML-like code within JavaScript.
- ✓ Helps differentiate files that use JSX from regular JavaScript files.
- ✓ Some developers prefer this for better clarity and organization.



Use .js:

- ✓ If you prefer simplicity and don't need to distinguish between JavaScript and JSX files.
- ✓ Few projects use .js for both regular JavaScript and React components to maintain consistency.

Vite expects files containing JSX to have a `jsx` or `tsx` extension. To use `js` for JSX, you can configure Vite by adding `esbuild: { loader: 'jsx' }` in `vite.config.js`. However, it's simpler and more consistent to use the `jsx` extension for React components.

Types of Components

Functional Components

- Simple JavaScript functions.
- Recommended and more popular
- Return JSX (HTML-like syntax).

Example:

```
const Header = () => {
  return <h1>Welcome to My
    App</h1>;
}
```



Class Components

- Older way of defining components, verbose.
- Use ES6 classes and include lifecycle methods

Example:

```
class Header extends React.Component
{
  render() {
    return <h1>Welcome to My
      App</h1>;
  }
}
```

Export and Import components

React applications are built with reusable components. To use a component in another file, you need to export it from one file and import it into another.

Two Ways to Export Components

1. Default Export

- Use this to export one main thing from a file.
- No curly braces are needed when importing.

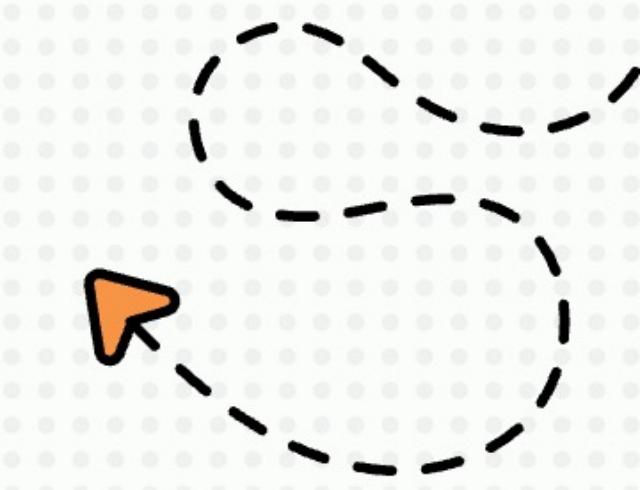
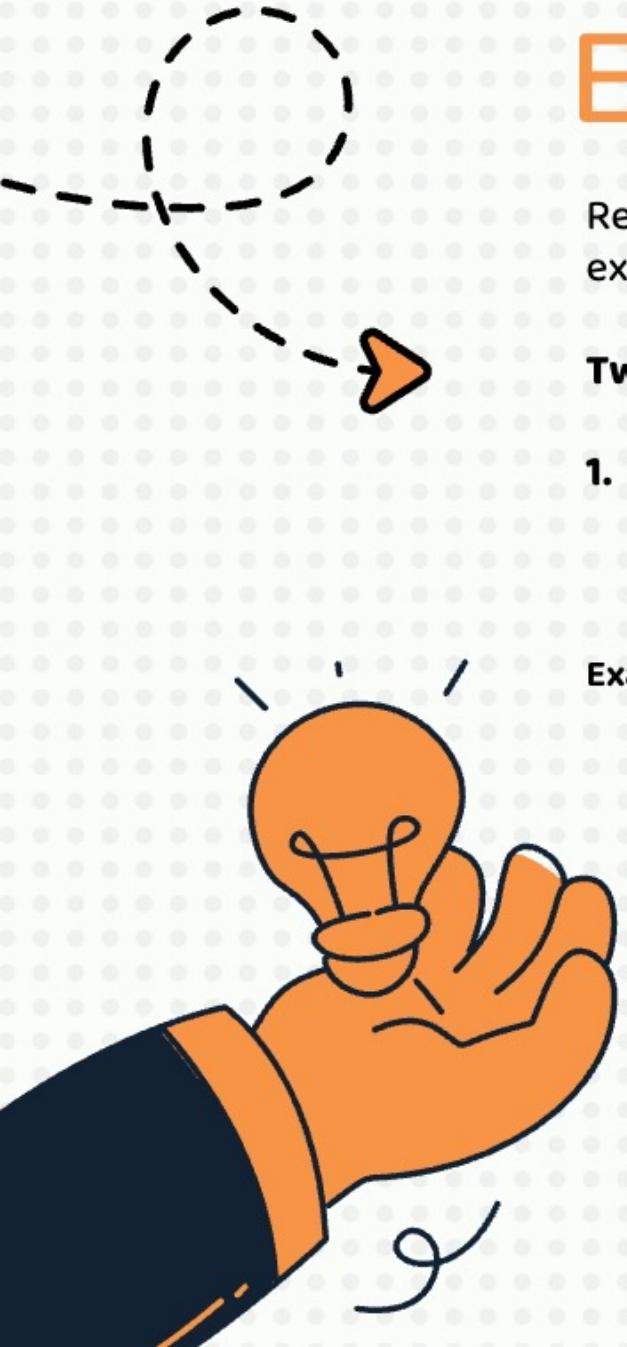
Example:

File: MyComponent.jsx

```
export default function MyComponent() {  
  return <h1>Hello, Default Export!</h1>;  
}
```

File: App.jsx

```
import MyComponent from './MyComponent';  
  
function App() {  
  return <MyComponent />;  
}
```



Export and Import components

2. Named Export

- Use this to export multiple components or functions from a file.
- Curly braces are required when importing.

Example:

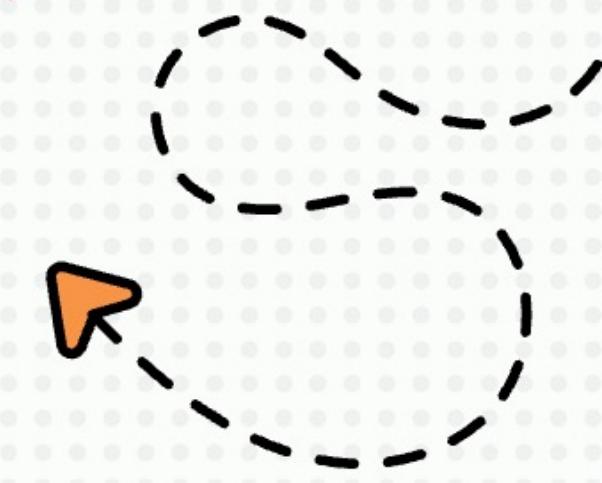
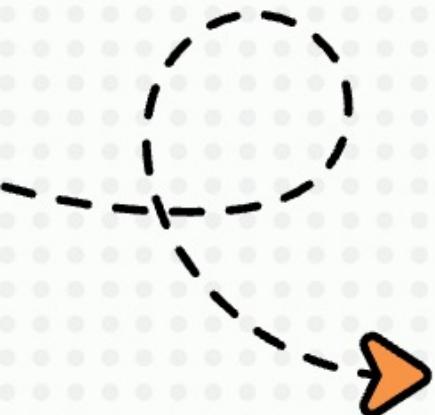
File: MyComponents.jsx

```
export default function Header() {  
  return <h1>Header Component</h1>;  
}
```

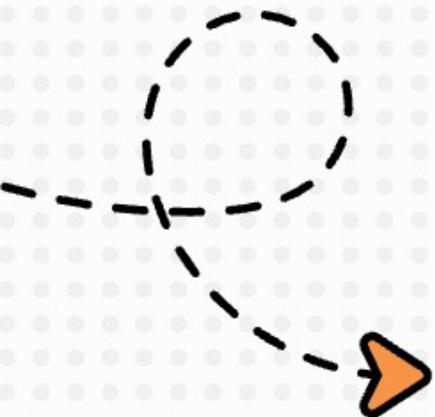
```
export function Footer() {  
  return <p>Footer Component</p>;  
}
```

File: App.jsx

```
import Header, {Footer} from './MyComponents';  
  
function App() {  
  return (  
    <>  
      <Header />  
      <Footer />  
    </>  
  );  
}
```



Export and Import components



Important Tips

1. **Alias Imports:** You can rename imports for better readability.

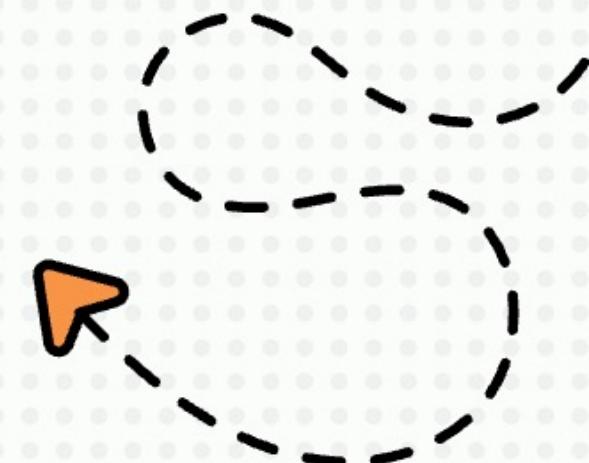
```
import MyHeader from './MyComponents'; // Alias for a default export
import { Footer as MyFooter } from './MyComponents'; // Alias for a named export
```

2. **Import All as an Object:** Import everything from a file. Useful for named exports.

```
import * as Components from './MyComponents';

function App() {
  return (
    <>
      <Components.Header />
      <Components.Footer />
    </>
  );
}
```

Remember: **Curly braces** are for **named exports**, not for default exports.

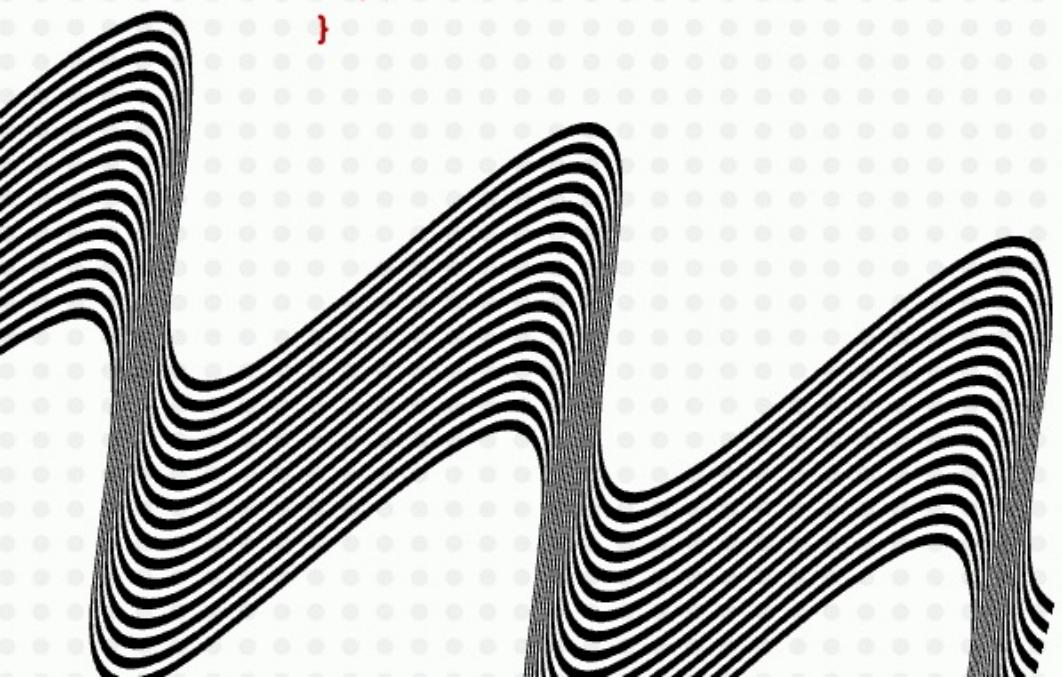


React Fragment

In React, every component must return a single JSX element. You can't directly return multiple sibling elements without wrapping them in a parent.

Example: This Won't Work

```
function App() {
  return (
    <h1>Title</h1>
    <p>Description</p>
  );
}
```



How Do We Fix This?

To fix this, we need to group these elements inside a parent element, such as a `<div>`.

Example: Using a `<div>`

```
function App() {
  return (
    <div>
      <h1>Title</h1>
      <p>Description</p>
    </div>
  );
}
```

Problem: This adds an unnecessary `<div>` to the DOM

React Fragment

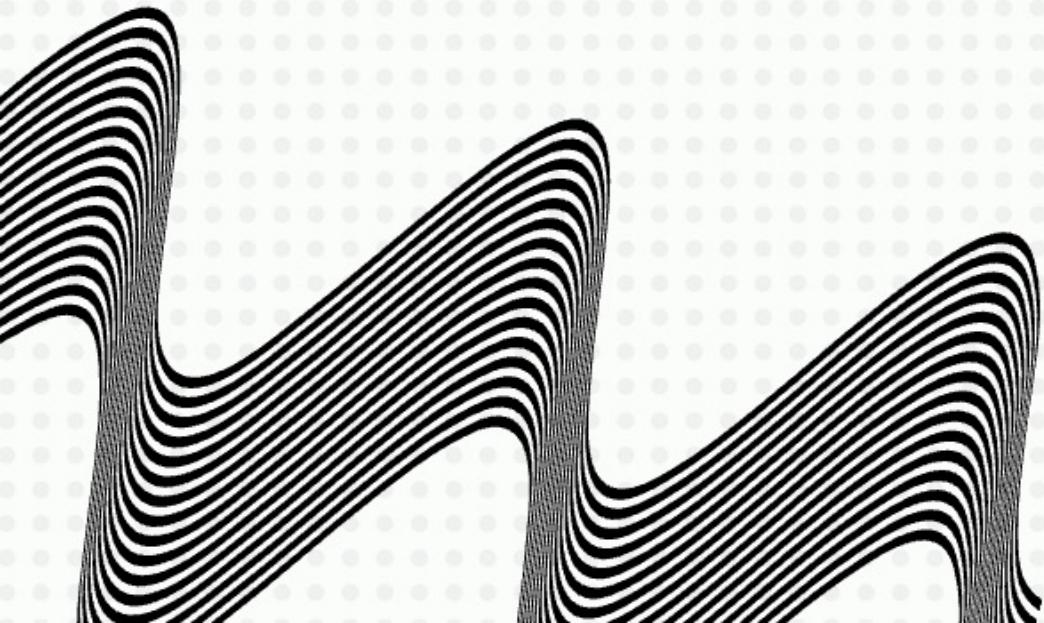
React Fragment allow you to group elements without adding extra DOM nodes.

Benefits of React Fragment

Cleaner DOM: Avoids unnecessary wrappers like <div>.

Improved Performance: Fewer DOM nodes mean faster rendering.

Better Layouts: Prevents extra tags from affecting your CSS or layout.



Example: Using a Fragment

```
function App() {
  return (
    <React.Fragment>
      <h1>Title</h1>
      <p>Description</p>
    </React.Fragment>
  );
}
```

Short Syntax for Fragments

Instead of <React.Fragment>, you can use short syntax (<> and </>):

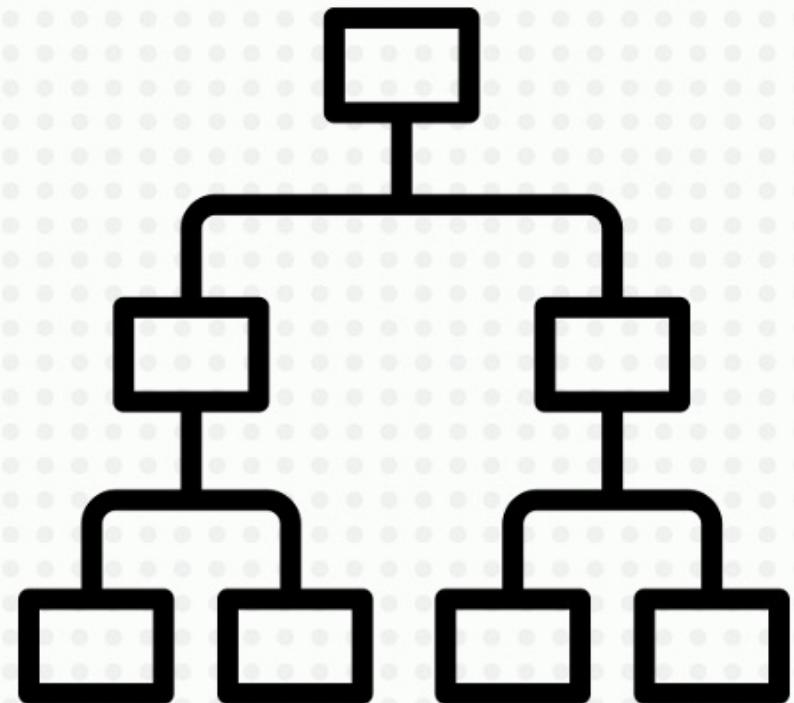
```
function App() {
  return (
    <>
      <h1>Title</h1>
      <p>Description</p>
    </>
  );
}
```

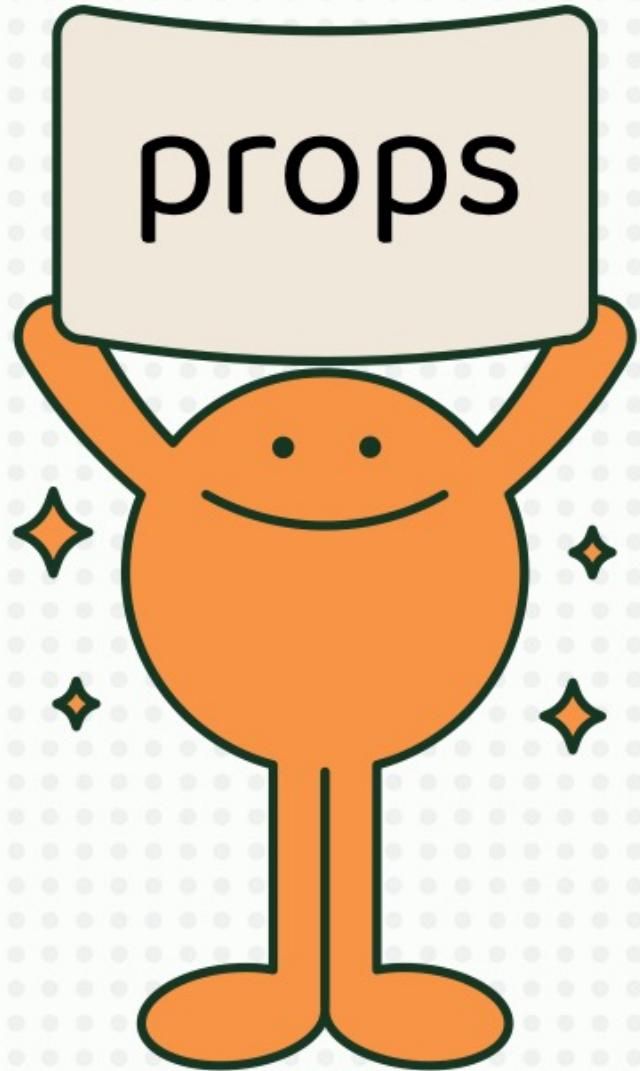
Parent and Child Components

- ✓ Components can contain other components.
- ✓ **Parent Component:** A component that includes other components.
- ✓ **Child Component:** A component nested inside another component.

Example:

```
function App() {  
  return (  
    <>  
    <Header />  
    <Footer />  
    </>  
  );  
}
```





What Are Props?

Props (short for “properties”) are how we pass data from one component to another. They make components reusable by allowing them to receive data and customize their behavior.

How Props Work

1. Props are like function arguments but for components.
2. They are passed from a parent component to a child component.
3. Props are read-only, meaning the child cannot modify them.

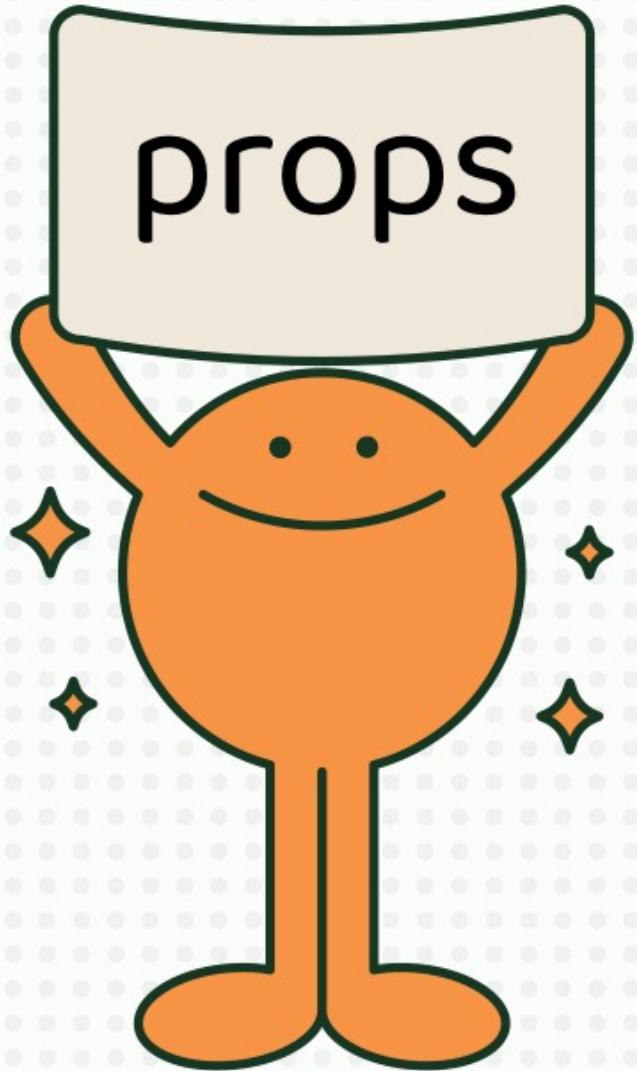
Example: Passing Props

Parent Component:

```
function App () {  
  return <Greeting name="John" />;  
}
```

Child Component:

```
function Greeting(props) {  
  return <h1>Hello, {props.name}!</h1>;  
}
```



Using Destructuring For Props

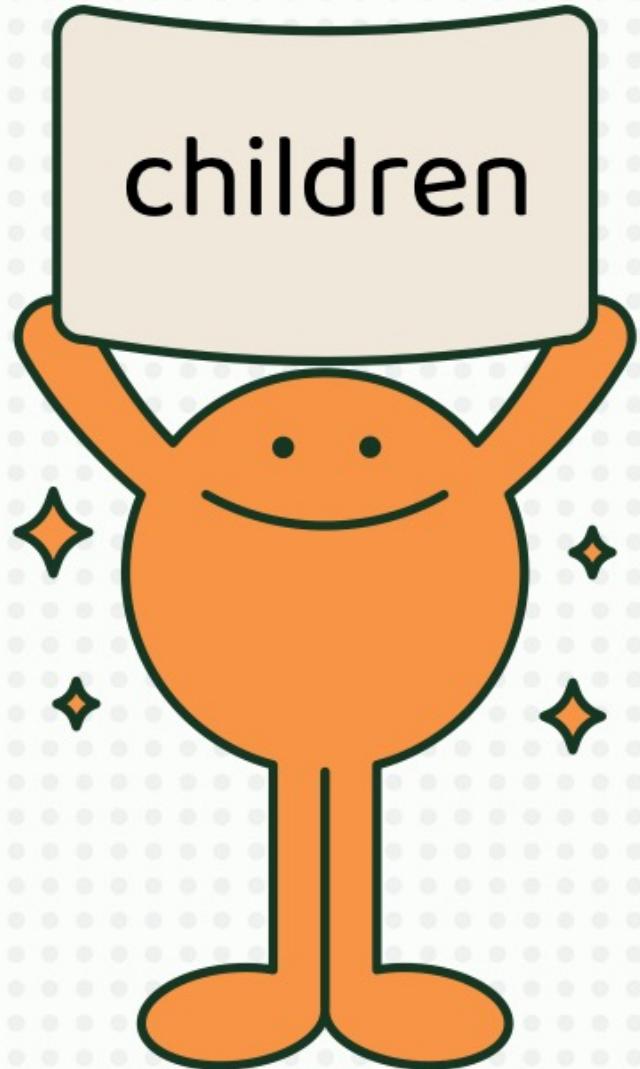
To simplify the syntax, you can destructure props:

```
function Greeting({ name }) {  
  return <h1>Hello, {name}!</h1>;  
}
```

Destructuring is a feature in JavaScript that allows you to extract values from arrays or properties from objects and assign them to variables in a clean and concise way.

Example:

```
const user = {  
  name: "John", age: 25, country: "USA" };  
  
// Destructuring  
const { name, age, country } = user;  
console.log(name); // Output: John  
console.log(age); // Output: 25  
console.log(country); // Output: UA
```



What Is children?

The `children` prop is a special prop in React that allows you to pass components, elements, or text as the content between the opening and closing tags of a component.

Example: Using children

Parent Component:

```
function App() {  
  return (  
    <Card>  
      <h2>Title</h2>  
      <p>This is some content inside the Card component.</p>  
    </Card>  
  );  
}
```

Child Component:

```
function Card({ children }) {  
  return <div className="card">{children}</div>;  
}
```

Key differences between **props** and **children**

Props	Children
Used to pass custom data or settings	Used to pass content between opening and closing tags
Explicitly named and accessed via <code>props.propName</code>	Automatically available as <code>props.children</code>
Example: <code><Component title="Hello" /></code>	Example: <code><Component>Content here</Component></code>



Props Are Immutable

Props **cannot be modified** by the child component. If a value needs to change, manage it in the parent component using **state**.

What is a **key** Prop?

The key prop is a special attribute used in React to uniquely identify elements in a list. It helps React efficiently update and render the UI when the list changes.

Why is the key Prop Important?

1. Improves Performance: React uses key to quickly identify which elements:
 - Need to be added.
 - Need to be removed.
 - Need to be updated.
2. Avoids Unexpected Bugs: Without a unique key, React may accidentally reuse or re-order elements incorrectly.

When to Use the key Prop?

You should use the key prop whenever rendering a list of elements using `.map()` or similar methods.

```
function App() {  
  return (  
    <ul>  
      {items.map((item) => <li key={item.id}>{item.name}</li>);}  
    </ul>  
  );  
}
```

Adding `key={item.id}` ensures React can uniquely identify each ``.



What is a **key** Prop?

Avoid Using Index as a Key (if possible):

Using index can cause issues if the list order changes. Use it only as a fallback.

Keys Must Be Stable:

The key value should not change between renders. This helps React maintain the correct element association.

What Happens Without Keys?

1. React will throw a warning **"Each child in a list should have a unique 'key' prop."**
2. React may recreate elements unnecessarily, which impacts performance.
3. UI updates may behave unpredictably, especially when adding or removing items.



Dynamic Components

- ✓ Dynamic components in React are components that are rendered dynamically based on certain conditions or loaded at runtime. They enable you to make your application flexible and interactive by rendering different components or loading them only when needed.
- ✓ **Conditional Rendering example**

```
function App () {
  const isLoggedIn = true;

  return (
    <div>
      {isLoggedIn ? <WelcomeUser /> : <Login />}
    </div>
  );
}
```

Renders `<WelcomeUser />` if logged in.
Renders `<Login />` if not logged in.

Dynamic Components

✓ Components using Data

```
const items = ["Item 1", "Item 2", "Item 3"];  
  
function App() {  
  return (  
    <ul>  
      {items.map((item, index) =>  
        <li key={index}>{item}</li>  
      )}  
    </ul>  
  );  
}
```

Generates `` for each item in the array.

Dynamic components make your React app more efficient and interactive by rendering only what is needed, when it is needed. Use them to optimize performance and create better user experiences.

Built-in components in React

React provides a set of built-in components that are essentially HTML-like elements but with additional capabilities to integrate with React's rendering logic. These are also known as DOM components.

Examples:

1. `<div>, , <p>` - Common layout elements.
2. `<input>, <button>, <form>` - Interactive elements.
3. `, <audio>, <video>` - Media elements.

Characteristics:

- ✓ Represent standard HTML elements.
- ✓ Start with lowercase letters (e.g., `<div>`).
- ✓ React directly translates them to DOM elements in the browser.

Whereas Custom components are user-defined, reusable building blocks for UI development. They are created by the developer to encapsulate logic, state, & styling. Start with uppercase letters.



Built-in components in React

Built-in components are used inside custom components to construct UI.

Example:

```
function Header() {  
  return (  
    <div className="header">  
      <h1>Welcome</h1>  
    </div>  
  );  
}
```



Key Differences Between **Built-in** and **Custom** components

Aspect	Built-in Components	Custom Components
Name Format	Lowercase (e.g., <div>)	Uppercase (e.g., <Header />)
Defined By	Part of React or the DOM	Created by the developer
Purpose	Render standard HTML elements	Build reusable, dynamic UI blocks
Props Support	Limited (e.g., className, id)	Fully customizable via props
Reusability	Not reusable in a React-specific way	Highly reusable and can encapsulate logic
Example Usage	<div>Content</div>	<Header title="Welcome" />

JSX vs HTML

- ✓ When working with React, developers write UI components using JSX (JavaScript XML), a syntax extension that looks similar to HTML.
- ✓ JSX allows us to combine JavaScript logic and UI representation seamlessly, enabling dynamic and reusable components.
- ✓ While JSX resembles HTML, it is not the same. It is a syntax sugar over React's createElement function and has some key differences and stricter rules.
- ✓ Understanding these differences is crucial for writing clean, bug-free React code and leveraging React's full potential in real-world projects.

1. **className instead of class**

JSX: Use className to define CSS classes.

```
<div className="container">Hello World</div>
```

HTML: Use class

```
<div class="container">Hello World</div>
```

Reason: class is a reserved keyword in JavaScript



JSX vs HTML

2. htmlFor instead of for

JSX: Use htmlFor to associate labels with form controls

```
<label htmlFor="username">Username</label>
<input id="username" type="text" />
```

HTML: Use for

```
<label for="username">Username</label>
```

Reason: For is a reserved keyword in JavaScript

3. Self-Closing Tags

JSX: Self-closing tags must always end with />.

```

```

HTML: Self-closing tags can omit the /.

```

```

Reason: JSX follows stricter syntax rules derived from XML, whereas HTML is more lenient.



JSX vs HTML

4. JavaScript Expressions

JSX: Use curly braces {} to embed JavaScript expressions.

```
<h1>{2 + 2}</h1>
```

HTML: No direct way to embed JavaScript.

5. Attributes and Event Handling

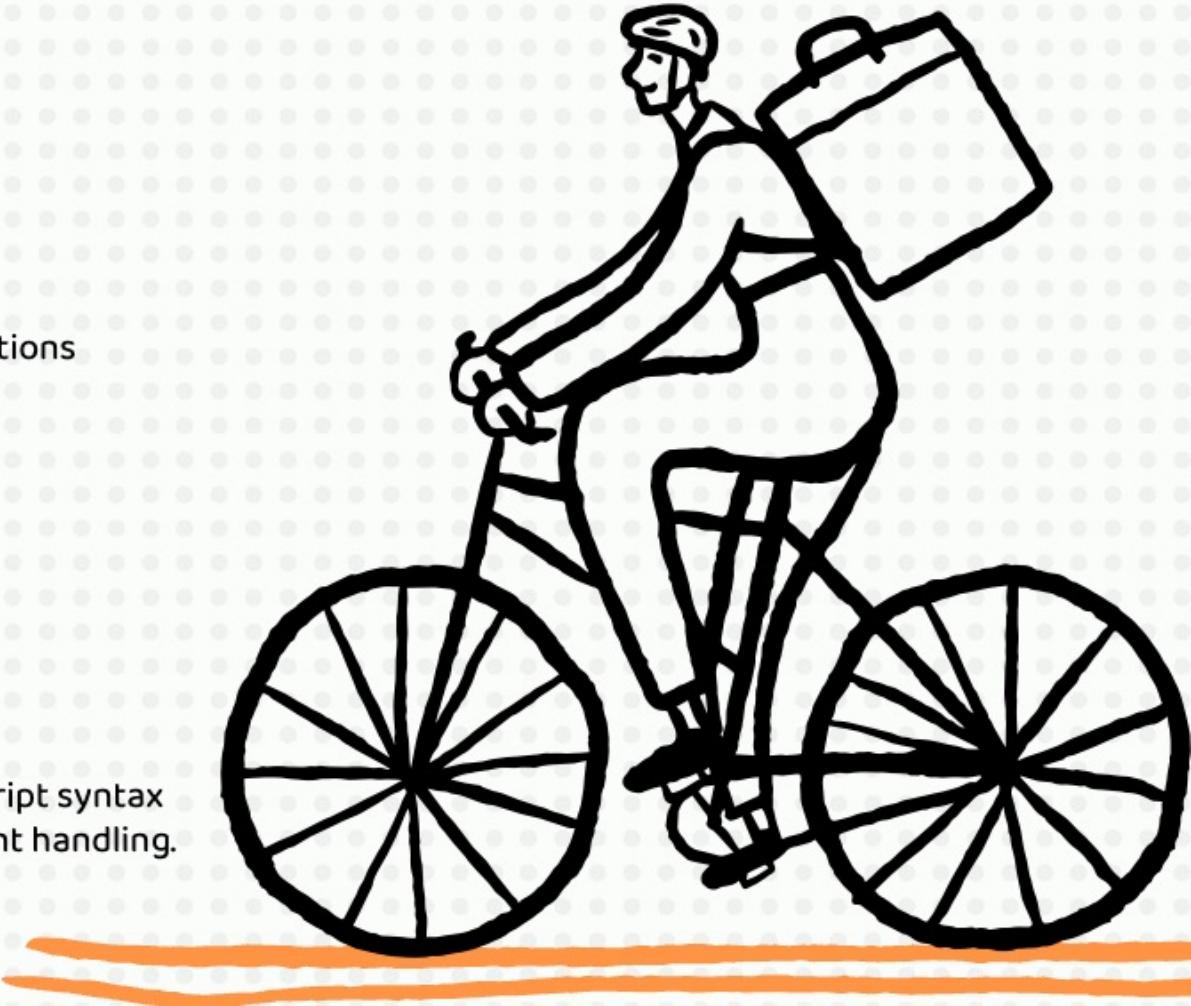
JSX uses camelCase for attributes, event handlers and requires passing functions

```
<input type="text" autoComplete="off" />
<button onClick={handleClick}>Click Me</button>
```

HTML uses lowercase for attributes, event handlers

```
<input type="text" autocomplete="off">
<button onclick="handleClick()">Click Me</button>
```

Reason: JSX being a JavaScript extension that integrates closely with JavaScript syntax and practices, while HTML follows its own set of rules for attributes and event handling.



6. style Attribute

JSX: Style is written as an object with camelCase properties.

```
<div style={{ backgroundColor: 'blue', color: 'white' }}>Hello</div>
```

HTML: Style is written as a string with CSS syntax.

```
<div style="background-color: blue; color: white;">Hello</div>
```

Reason: The difference in how style is written in JSX and HTML is due to JSX being JavaScript-based, while HTML is purely a markup language.

7. Comments

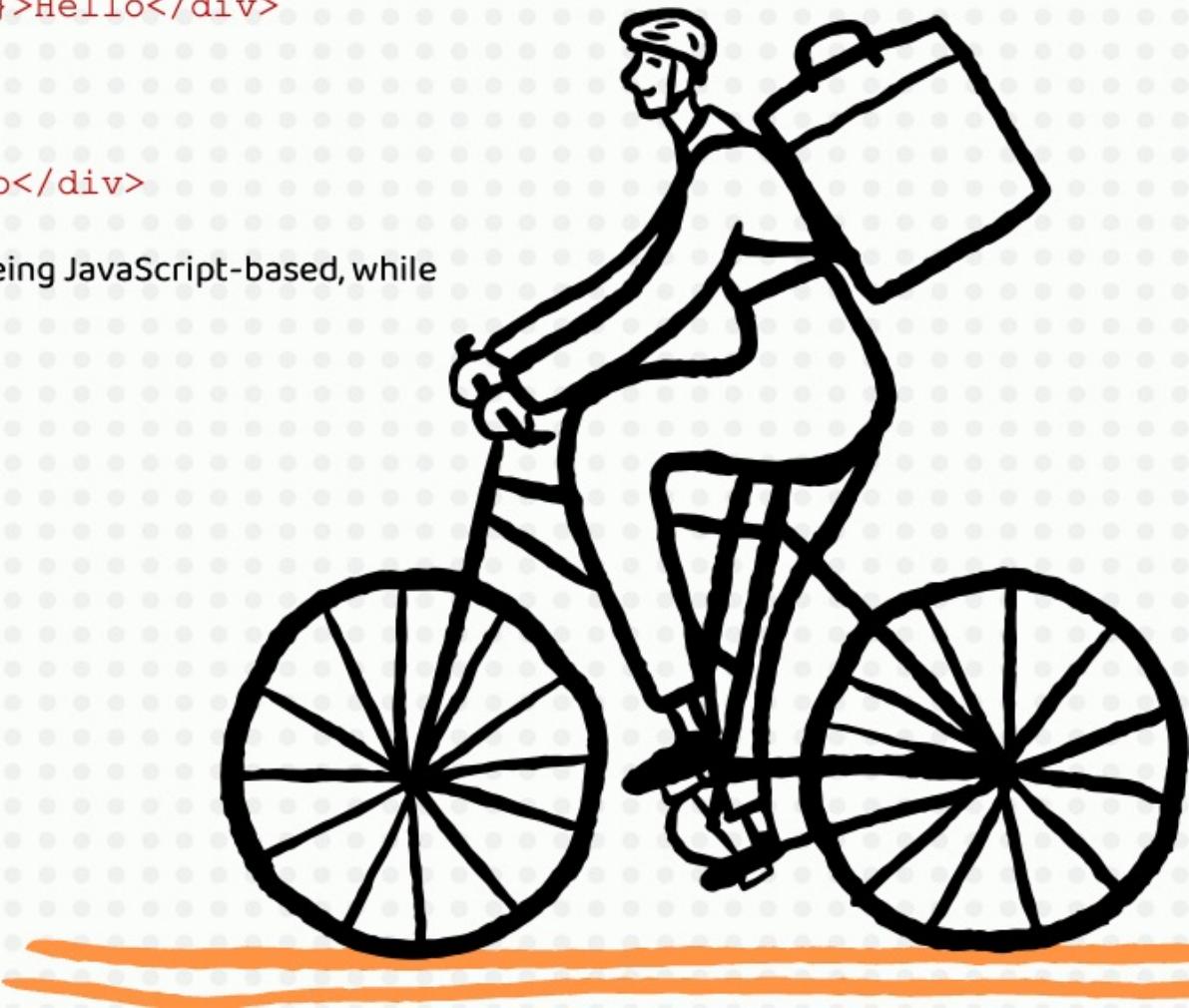
JSX: Use `/* */` for comments.

```
<div> /* This is a comment */ </div>
```

HTML: Use `<!---->`.

```
<div> <!-- This is a comment --> </div>
```

Reason: JSX is not HTML but a syntax extension of JavaScript, so it uses JavaScript-compatible commenting syntax.



8. Boolean Attributes

In plain HTML, boolean attributes are true if they are present and false if they are absent. You don't need to explicitly set their value.

```
<!-- 'checked' is true because it's present -->  
<input type="checkbox" checked>  
  
<!-- 'checked' is false because it's absent -->  
<input type="checkbox">
```

In JSX, you can directly assign true or false to boolean attributes. This is more explicit than HTML.

```
<input type="checkbox" checked={true} />
```

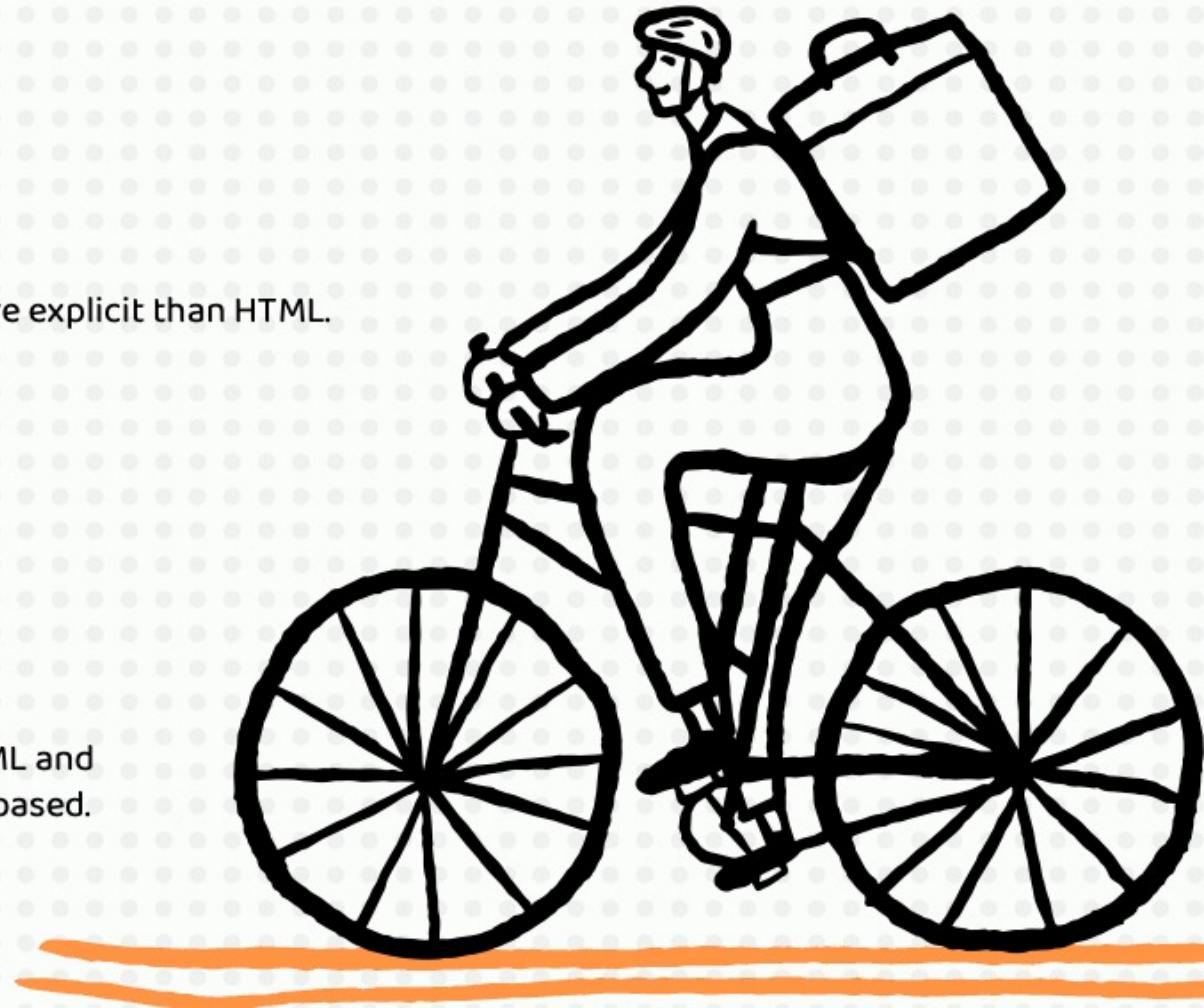
In JSX, if you don't specify a value, it is assumed to be true:

```
<input type="checkbox" checked />
```

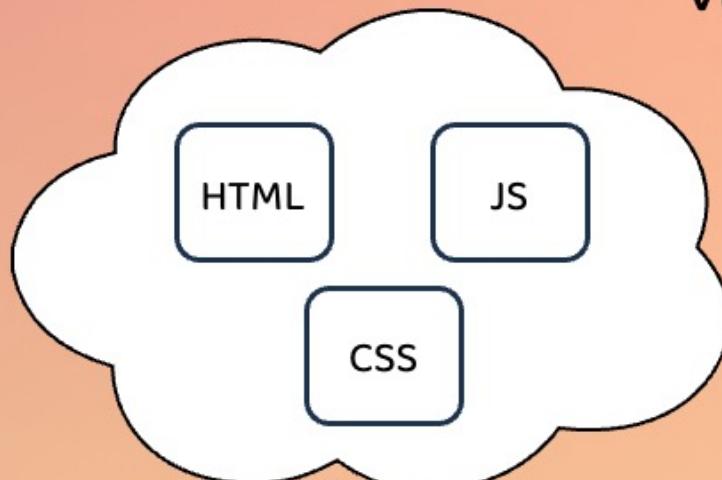
This is equivalent to:

```
<input type="checkbox" checked={true} />
```

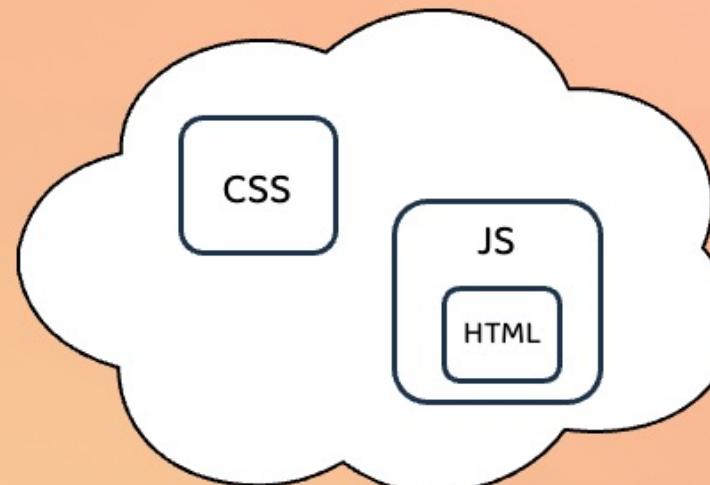
Reason: The differences between how boolean attributes work in plain HTML and JSX stem from the fact that HTML is markup-based, while JSX is JavaScript-based.



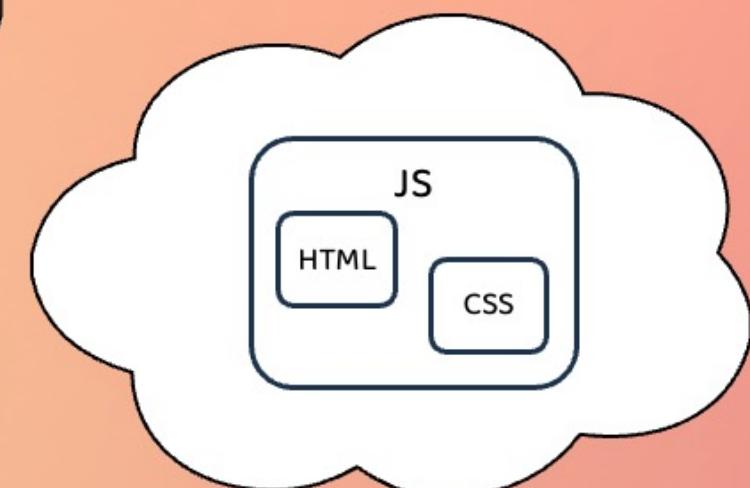
Various approaches to apply **CSS**



Old style of web development



Modern web development
with HTML inside JS (JSX)



Modern web development
with HTML, CSS inside JS (JSX)

When working with React, we have an option to include CSS code either in JSX or in a separate file

Styling React Apps

- ✓ Building functional apps is great, but making them visually appealing ensures users enjoy using them.
- ✓ In React, styling is achieved with CSS, which differs from HTML and JavaScript.
- ✓ We'll explore three popular methods for styling React apps

01

CSS files and class names

02

CSS Modules

03

Styled-components

Styling React Apps using CSS files and class names

Using CSS files and class names is the simplest and most traditional approach to styling React applications. You write your styles in a .css file and apply them to elements using the `className` attribute.

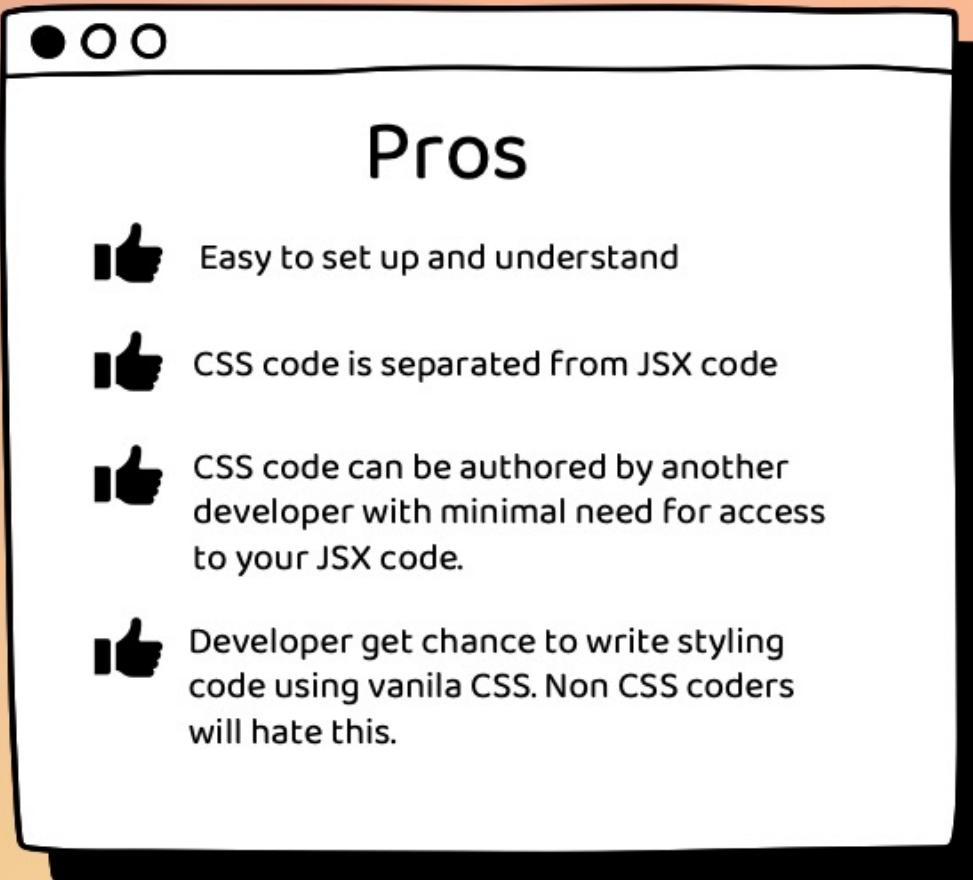
1. Create a CSS File: Write your styles in a .css file (e.g., styles.css):

```
.heading {  
    font-size: 24px;  
    color: #333;  
}  
  
.button {  
    background-color: #007bff;  
    color: white;  
    padding: 10px 20px;  
    border: none;  
    border-radius: 5px;  
}
```

2. Import the CSS File and Apply Classes to Elements

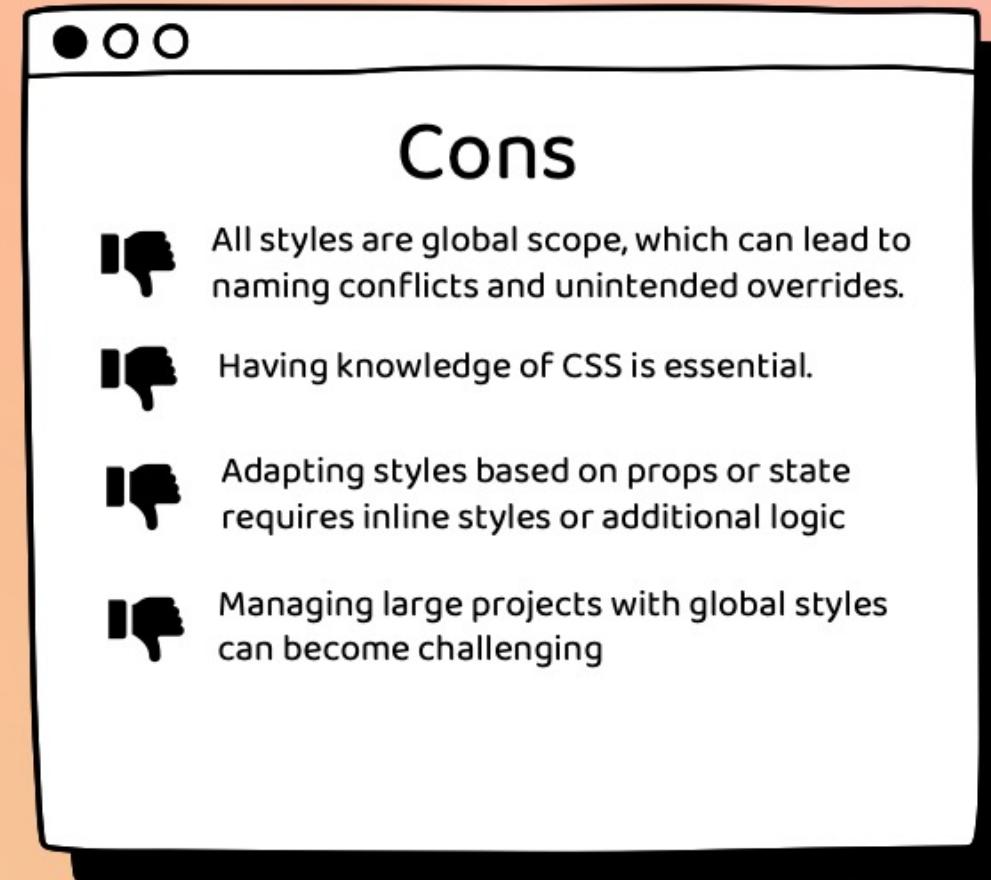
```
import './styles.css';  
  
function App() {  
    return (  
        <div>  
            <h1 className="heading">Welcome to React</h1>  
            <button className="button">Click Me</button>  
        </div>  
    );  
}  
export default App;
```

Styling React Apps using CSS files and class names



Pros

- + | Easy to set up and understand
- + | CSS code is separated from JSX code
- + | CSS code can be authored by another developer with minimal need for access to your JSX code.
- + | Developers get chance to write styling code using vanilla CSS. Non CSS coders will hate this.



Cons

- | All styles are global scope, which can lead to naming conflicts and unintended overrides.
- | Having knowledge of CSS is essential.
- | Adapting styles based on props or state requires inline styles or additional logic
- | Managing large projects with global styles can become challenging

Styling React Apps with Inline Styles

Inline styles in React involve applying styles directly to elements using the style attribute. Styles are written as JavaScript objects, making them dynamic and easy to manipulate programmatically.

Apply Styles with the style Attribute

```
function App() {  
  
  const buttonStyle = {  
    backgroundColor: '#007bff',  
    color: 'white',  
    padding: '10px 20px',  
    border: 'none',  
    borderRadius: '5px',  
  };  
  
  return (  
    <div>  
      <h1 style={{fontSize: '24px', color: '#333'}}>Welcome to React</h1>  
      <button style={buttonStyle}>Click Me</button>  
    </div>  
  );  
}  
export default App;
```

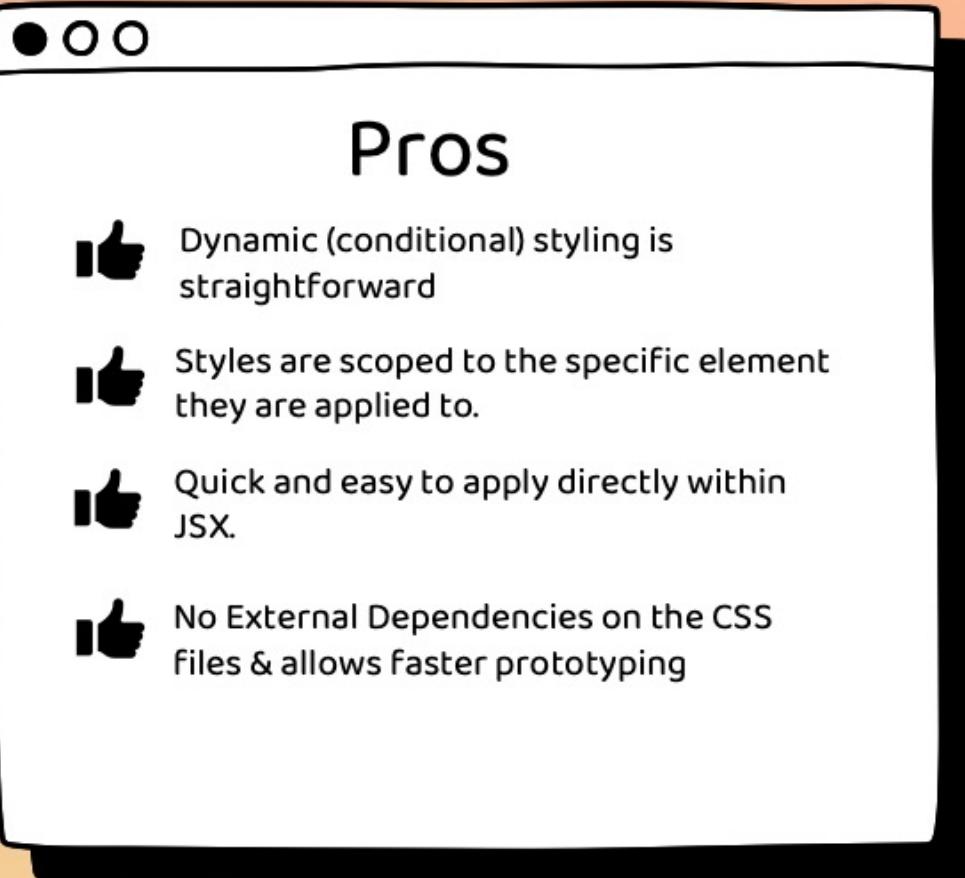
Styling React Apps with Inline Styles

Update styles dynamically using conditions, props or state:

```
function App() {  
  
  const isActive = Math.random() > 0.5; // Dummy condition to determine active/inactive  
  
  const buttonStyle = {  
    'background-color': isActive ? '#28a745' : '#007bff',  
    color: 'white',  
    padding: '10px 20px',  
    border: 'none',  
    'border-radius': '5px',  
  };  
  
  return (  
    <button style={buttonStyle}>  
      {isActive ? 'Active' : 'Inactive'}  
    </button>  
  );  
  
  export default App;
```

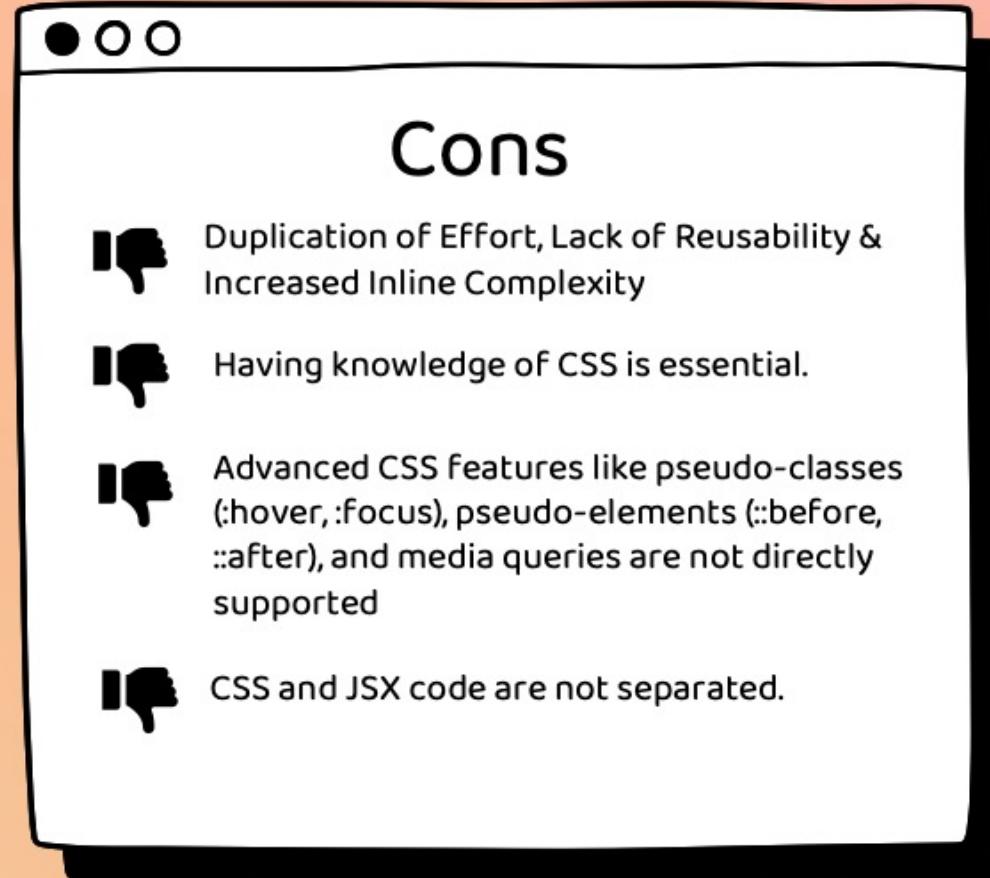
- To use native CSS properties like `background-color` instead of the `camelCase` convention, you can surround them with single quotes in the JSX style object. However, this approach is not widely adopted by developers.
- For single-word CSS properties, the syntax remains consistent between native CSS and JSX style objects.

Styling React Apps with Inline Styles



Pros

- Dynamic (conditional) styling is straightforward
- Styles are scoped to the specific element they are applied to.
- Quick and easy to apply directly within JSX.
- No External Dependencies on the CSS files & allows faster prototyping



Cons

- Duplication of Effort, Lack of Reusability & Increased Inline Complexity
- Having knowledge of CSS is essential.
- Advanced CSS features like pseudo-classes (:hover, :focus), pseudo-elements (:before, ::after), and media queries are not directly supported
- CSS and JSX code are not separated.

Using Template Literals (\${}) to combine static and dynamic styles

In JavaScript and React, you can combine static and dynamic styles using template literals (\${}) inside the className or style attributes.

```
const Card = ({ hasShadow, isRounded }) => {
  return (
    <div className={`card ${hasShadow ? "shadow" : ""} ${isRounded ? "rounded" : ""}`}>
      I am a Card
    </div>
  );
};
```

Explanation:

- "card" → Always applied.
- "shadow" → Applied only if hasShadow is true.
- "rounded" → Applied only if isRounded is true.

You can use template literals (\${}) for **inline styles** too.

```
<div style={{
  width: `${isLarge ? "200px" : "100px"} `,
  height: "100px",
  backgroundColor: isLarge ? "blue" : "gray",
}}>
  I am a Box
</div>
```

Styling React Apps with CSS Modules

What are CSS Modules?

- ✓ CSS Modules are a way to scope CSS styles locally to a component in React.
- ✓ They prevent global conflicts by generating unique class names automatically.
- ✓ Each component has its own styles, making it more maintainable.

The CSS file must have the .module.css or .module.scss extension for build tool like Vite to recognize it as a CSS Module

How to Use CSS Modules in React?

Step 1: Create a CSS Module File

```
/* Card.module.css */  
  
.card {  
  background-color: #f5f5f5;  
  padding: 20px;  
  border-radius: 10px;  
}  
  
.shadow {  
  box-shadow: 0px 4px 8px rgba(0, 0, 0, 0.2);  
}
```

Step 2: Import It in a React Component

```
import styles from './Card.module.css';  
  
const Card = () => {  
  return <div className={`${styles.card}  
    ${styles.shadow}`}>  
    I am a Card  
  </div>;  
};
```

Styling React Apps with CSS Modules

How Does CSS Modules Work?

CSS Modules rename class names automatically to avoid conflicts.

```
/* Card.module.css */
```

```
.card {  
  background-color: #f5f5f5;  
  padding: 20px;  
  border-radius: 10px;  
}
```

compiles to something like

```
.Card_module__card_abc123 {  
  background-color: #f5f5f5;  
  padding: 20px;  
  border-radius: 10px;  
}
```

Use `className={styles.className}` instead of Strings

```
<div className="card">Incorrect Usage</div>  
<div className={styles.card}>Correct Usage</div>
```

✗ Won't work

✓ Works with CSS Modules

Styling React Apps with CSS Modules

Dynamic Styling

We can conditionally apply styles in React with CSS Modules

```
/* Button.module.css */

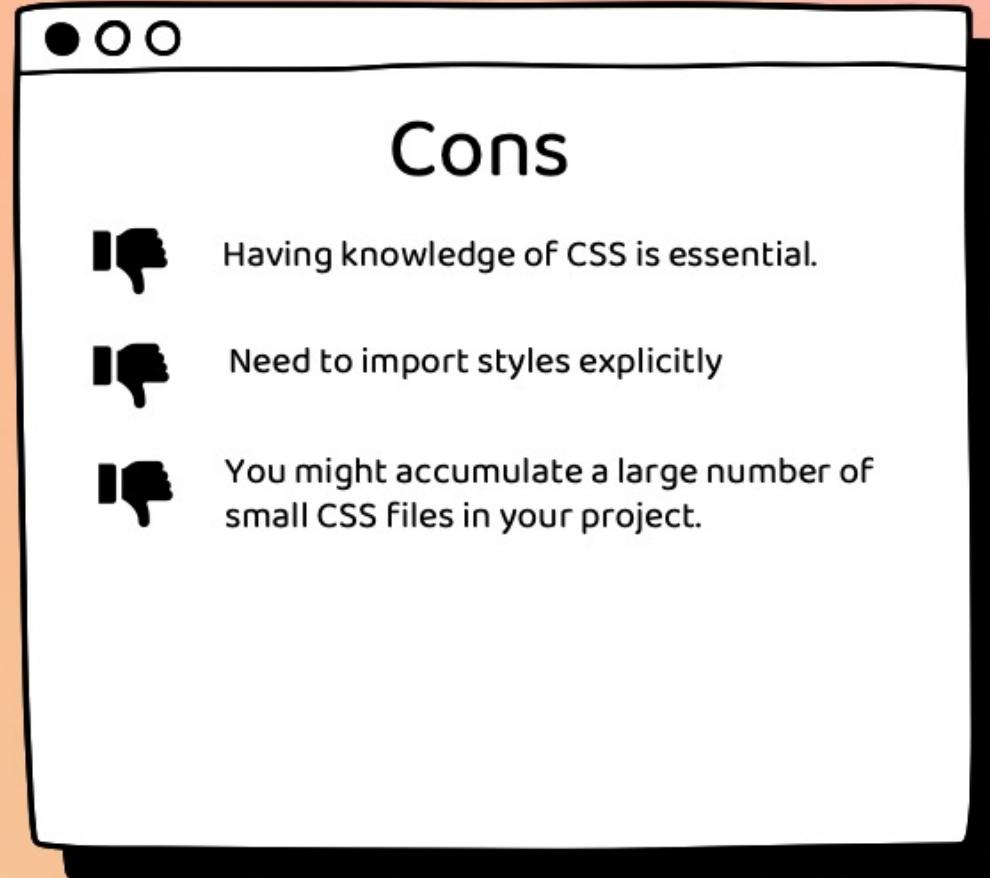
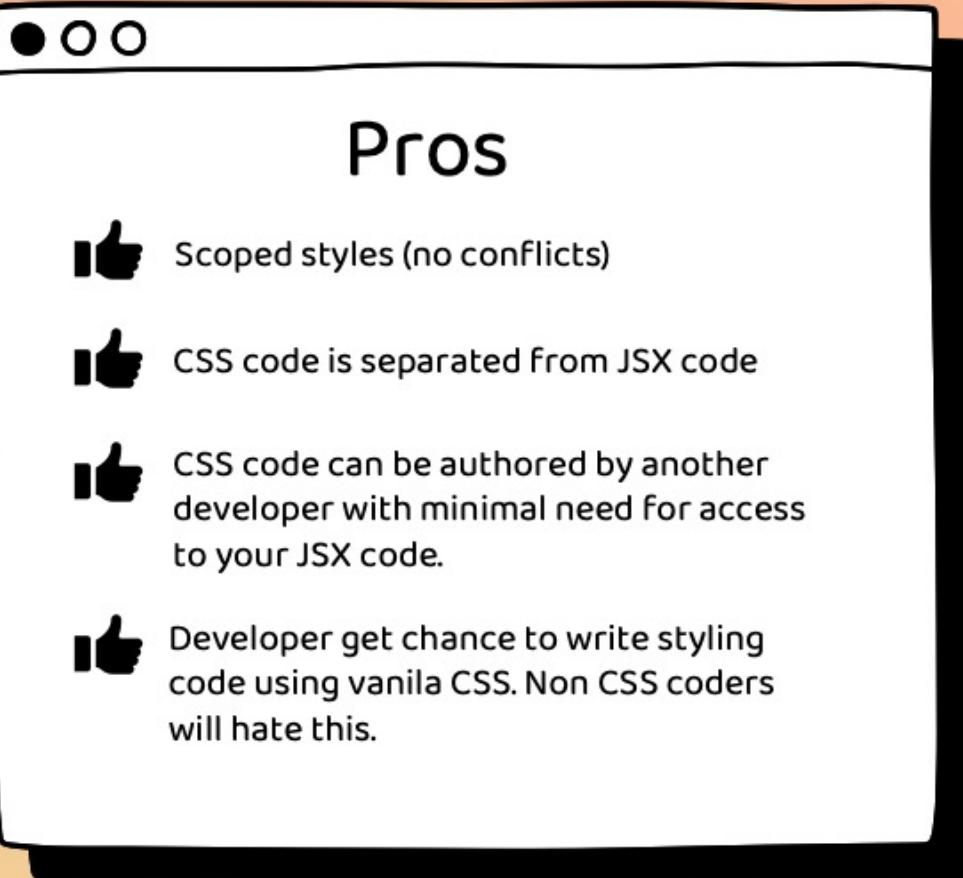
.primary {
  background-color: blue;
  color: white;
}

.secondary {
  background-color: gray;
  color: black;
}
```

```
import styles from './Button.module.css';

const Button = ({ primary }) => {
  return (
    <button className={primary ? styles.primary :
      styles.secondary}>
      Click Me
    </button>
  );
};
```

Styling React Apps with CSS Modules



Styling React Apps with Styled-Components

What are Styled-Components?

- ✓ Styled-Components is a library for styling React components using CSS-in-JS.
- ✓ It allows you to write CSS directly inside JavaScript files.
- ✓ Styles are scoped to the component, avoiding global conflicts.

Creating a Styled Button

```
import styled from 'styled-components';

const Button = styled.button`
  background-color: blue;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
  cursor: pointer;
`;

const App = () => {
  return <Button>Click Me</Button>;
};

export default App;
```

Explanation:

- styled.button → Creates a styled <button>.
- CSS is written inside backticks (`) like normal CSS.
- The component <Button> can be used just like any React component.

Setting Up Styled-Components

```
npm install styled-components
```

Styling React Apps with Styled-Components

Styled-components allow **conditional styling using props**

Changing Button Color Dynamically

```
const Button = styled.button`  
  background-color: ${props => (props.$primary ? "blue" : "gray")};  
  color: white;  
  padding: 10px 20px;  
  border: none;  
  border-radius: 5px;  
  cursor: pointer;  
`;  
  
const App = () => {  
  return (  
    <>  
      <Button $primary>Primary Button</Button>  
      <Button>Secondary Button</Button>  
    </>  
  );  
};
```

Styling React Apps with Styled-Components

Pseudo-Selectors in Styled-Components

```
import styled from 'styled-components';

const Button = styled.button`
  background-color: blue;
  color: white;
  cursor: pointer;

  &:hover {
    background-color: darkblue;
  }

  &:focus {
    outline: 2px solid white;
  }
`;

const App = () => {
  return <Button>Hover & Focus Me</Button>;
};

export default App;
```

Explanation:

- &:hover → Changes background color when hovered.
- &:focus → Adds an outline when clicked or focused.

Styling React Apps with Styled-Components

Nested Rules in Styled-Components

```
const Card = styled.div`  
  background-color: white;  
  border-radius: 10px;
```

```
  & h2 {  
    color: blue;  
  }  
  
& p {  
  color: gray;  
}  
`;
```

```
const App = () => {  
  return (  
    <Card>  
      <h2>Card Title</h2>  
      <p>This is some card content.</p>  
    </Card>  
  );  
};  
  
export default App;
```

Explanation:

h2 & p inside Card will automatically get styles **without extra class names.**

Styling React Apps with Styled-Components

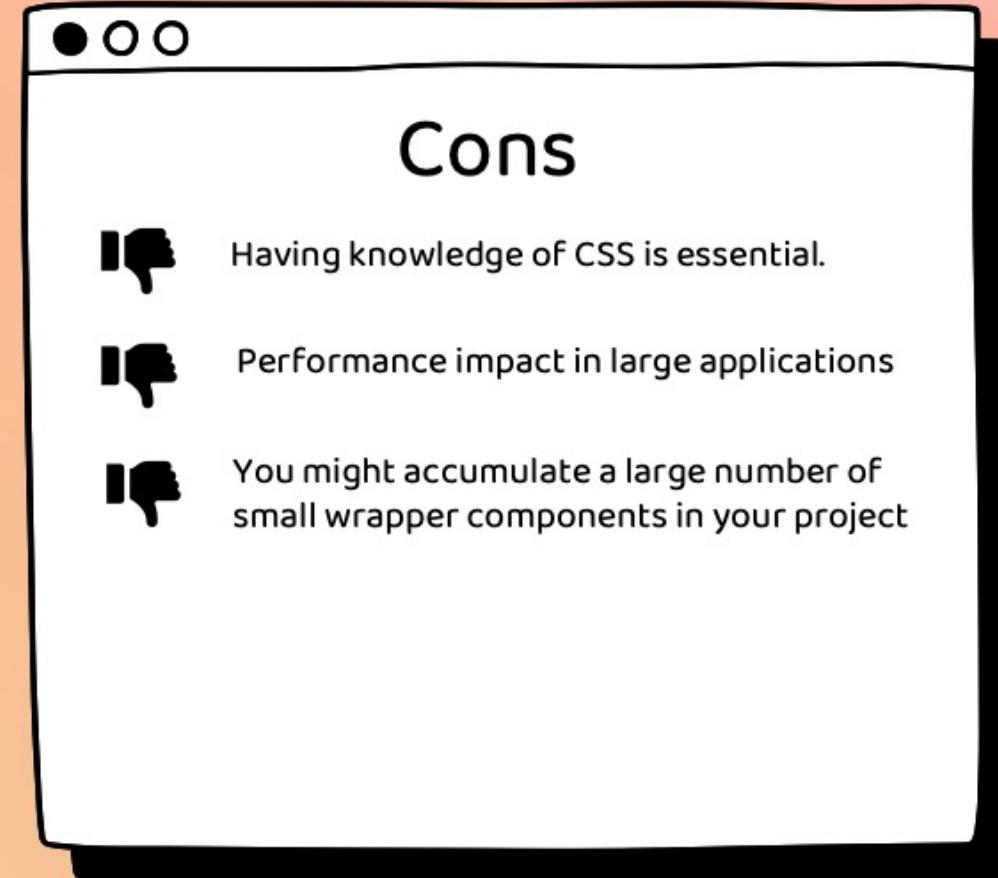
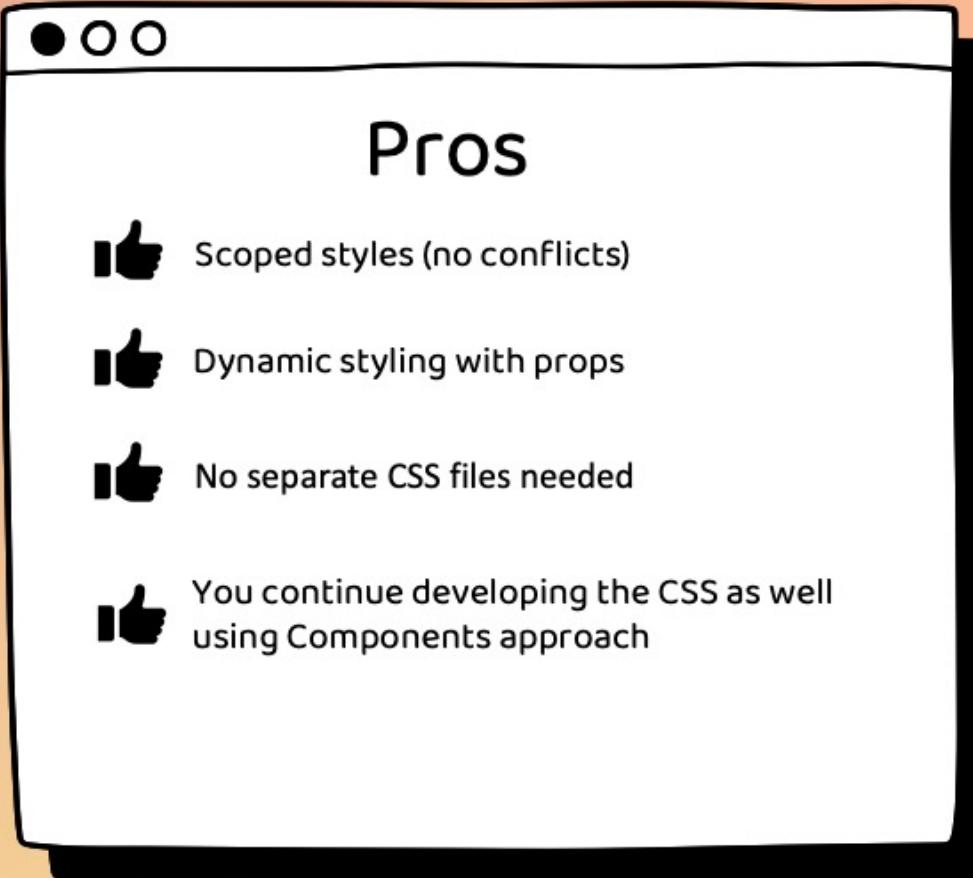
Media Queries in Styled-Components

```
const ResponsiveButton = styled.button`  
background-color: blue;  
color: white;  
padding: 10px 20px;  
border: none;  
border-radius: 5px;  
cursor: pointer;  
font-size: 16px;  
  
@media (max-width: 600px) {  
background-color: red;  
font-size: 12px;  
}  
;  
  
const App = () => {  
return <ResponsiveButton>Resize Me</ResponsiveButton>;  
};  
  
export default App;
```

Explanation:

- Default button is blue with font-size 16px.
- On screens less than 600px, the button turns red with font-size 12px.

Styling React Apps with Styled-Components



Using Bootstrap in React

eazy
bytes

What is Bootstrap?

- ✓ Bootstrap is a popular CSS framework that helps design responsive and modern web pages
- ✓ It provides predefined styles and components like buttons, grids, and forms
- ✓ Bootstrap can be used in React to speed up development and improve UI design

How to Install Bootstrap in React?

Step 1: Run the below npm command

```
npm install bootstrap
```

Step 2: Then, import Bootstrap CSS in App.jsx or main.jsx using below statement,

```
import 'bootstrap/dist/css/bootstrap.min.css';
```



- ✓ Bootstrap makes styling React apps easy.
- ✓ Use classes like btn, container, row, col for layout and styling.
- ✓ Bootstrap components like buttons, forms, navbar are easy to integrate
- ✓ Bootstrap is mobile-friendly and responsive by default.

Using Bootstrap in React

Buttons example

```
<button class="btn btn-primary">Primary Button</button>
<button class="btn btn-secondary">Secondary</button>
<button class="btn btn-success">Success</button>
<button class="btn btn-danger">Danger</button>
```

- ✓ btn → Base button class
- ✓ btn-primary → Blue button
- ✓ btn-success → Green button
- ✓ btn-danger → Red button

Alerts example

```
<div class="alert alert-warning">This is a warning alert!</div>
<div class="alert alert-success">Success! Your action was completed.</div>
<div class="alert alert-danger">Error! Something went wrong.</div>
```

- ✓ alert → Base alert class
- ✓ alert-warning → Yellow warning box
- ✓ alert-danger → Red error box

Using Bootstrap in React

Cards example

```
<div class="card" style="width: 18rem;">
  
  <div class="card-body">
    <h5 class="card-title">Card Title</h5>
    <p class="card-text">This is a simple Bootstrap card.</p>
    <a href="#" class="btn btn-primary">Read More</a>
  </div>
</div>
```

- ✓ card → Creates a card
- ✓ card-img-top → Image at the top
- ✓ card-body → Content inside the card
- ✓ card-title → Card heading
- ✓ btn btn-primary → Button inside the card

Grid layout example

```
<div class="row gy-3 justify-content-center mt-3 text-center">
  <div class="col-3 border p-3 bg-warning text-dark">Column 1</div>
  <div class="col-3 border p-3 bg-danger text-white">Column 2</div>
  <div class="col-3 border p-3 bg-info text-dark">Column 3</div>
</div>
```

- ✓ row: Creates a grid row
- ✓ gy-3: Adds vertical spacing (1rem) between rows
- ✓ justify-content-center: Horizontally centers the columns
- ✓ mt-3: Adds a top margin of 1rem
- ✓ text-center: Centers text inside the columns
- ✓ col-3: Each column occupies 3/12 (25%) of the row width
- ✓ border: Adds a border around the column
- ✓ p-3: Adds padding (1rem) inside the column
- ✓ text-dark, text-white: Adjusts text color for contrast

Using Bootstrap in React

eazy
bytes

Forms example

```
<form>
  <div class="mb-3">
    <label for="email" class="form-label">Email address</label>
    <input type="email" class="form-control" id="email" placeholder="Enter email">
  </div>
  <div class="mb-3">
    <label for="password" class="form-label">Password</label>
    <input type="password" class="form-control" id="password" placeholder="Enter
      password">
  </div>
  <button type="submit" class="btn btn-success">Submit</button>
</form>
```

- ✓ **form-control** → Styles input fields
- ✓ **mb-3** → Adds bottom margin (spacing)
- ✓ **form-label** → Styles the label

Using Bootstrap in React

eazy
bytes

How to customize Bootstrap provided components ?

Option 1: Use the same CSS class name and make CSS changes inside it

```
.btn-primary {  
    background-color: #a7b621 !important;  
}
```

!important ensures your custom styles override Bootstrap.

Option 2: Using SCSS to Customize Bootstrap

- Install Sass using:`npm install sass`
- Create a `custom.scss` file and override Bootstrap variables

```
$primary: purple; // Change primary color to purple
```

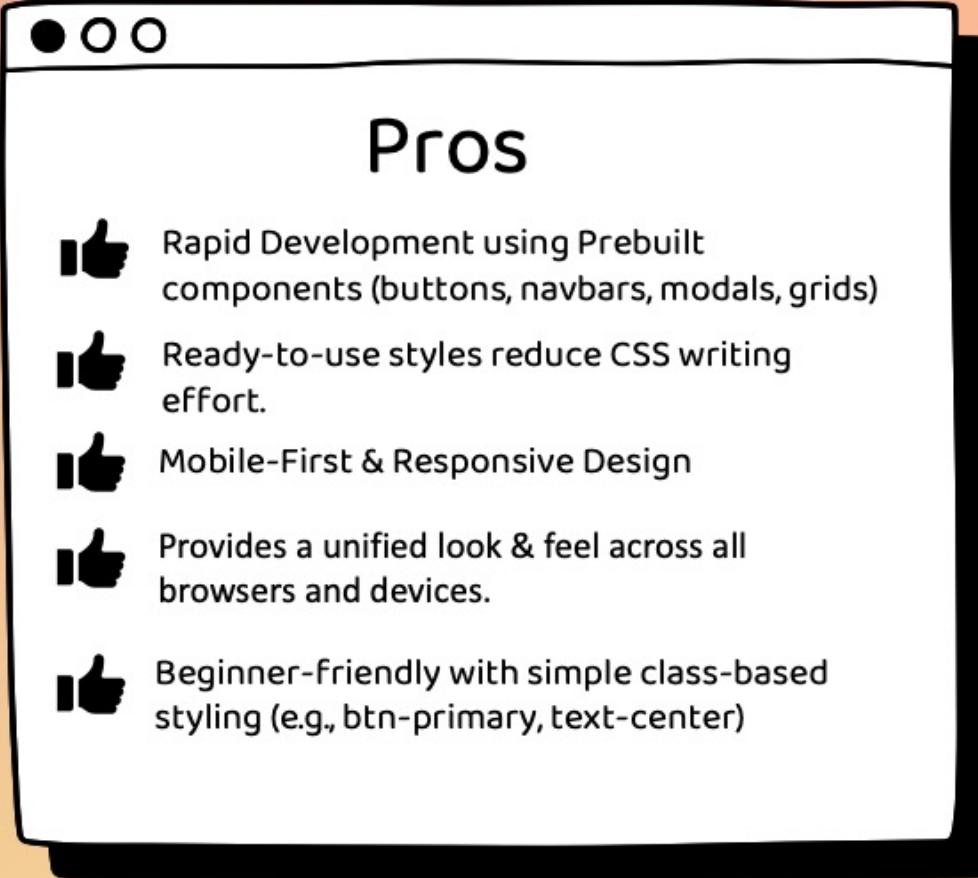
```
@import "bootstrap/scss/bootstrap"; // Import Bootstrap after overriding variables
```

\$primary modifies all `.btn-primary` buttons.

- Import `custom.scss` in your project

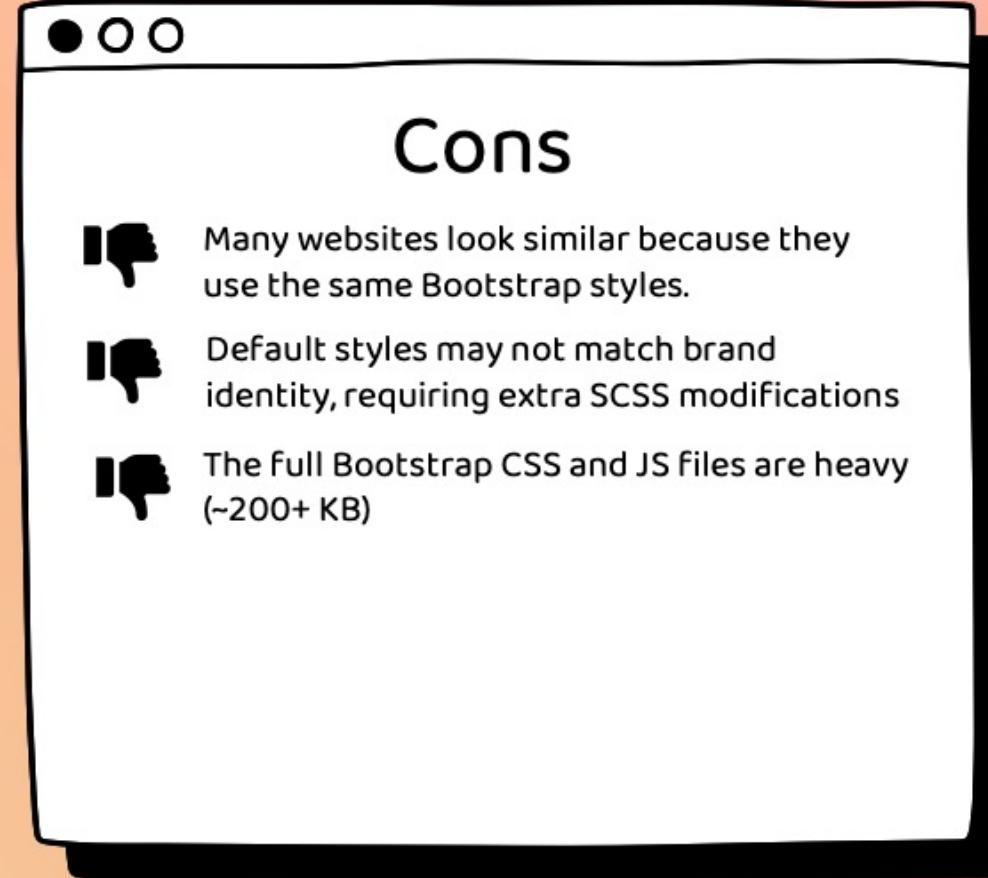


Using Bootstrap in React



Pros

-  Rapid Development using Prebuilt components (buttons, navbars, modals, grids)
-  Ready-to-use styles reduce CSS writing effort.
-  Mobile-First & Responsive Design
-  Provides a unified look & feel across all browsers and devices.
-  Beginner-friendly with simple class-based styling (e.g., btn-primary, text-center)



Cons

-  Many websites look similar because they use the same Bootstrap styles.
-  Default styles may not match brand identity, requiring extra SCSS modifications
-  The full Bootstrap CSS and JS files are heavy (~200+ KB)

Using Tailwind CSS in React

eazy
bytes

What is Tailwind CSS ?

- ✓ A utility-first CSS framework for rapid UI development.
- ✓ Provides predefined utility classes instead of predefined components.
- ✓ Allows developers to directly style elements in HTML without writing CSS from scratch.

How to setup Tailwind CSS using Vite ?

Step 1: Install tailwindcss and @tailwindcss/vite via npm

```
npm install tailwindcss @tailwindcss/vite
```

Step 2: Add the @tailwindcss/vite plugin to your Vite configuration.

```
import { defineConfig } from 'vite'
import tailwindcss from '@tailwindcss/vite'
export default defineConfig({
  plugins: [
    tailwindcss(),
  ],
})
```



Using Tailwind CSS in React

eazy
bytes

Step 3: Import Tailwind CSS

Add an @import to your CSS file that imports Tailwind CSS.

```
@import "tailwindcss";
```

Step 4: Start using Tailwind in your HTML

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="/src/styles.css" rel="stylesheet">
</head>
<body>
  <h1 class="text-3xl font-bold underline">
    Hello world!
  </h1>
</body>
</html>
```

For more details, check the below link,

<https://tailwindcss.com/docs/installation/using-vite>

Using Tailwind CSS in React

eazy
bytes

Typography (Text Styling) example

```
<p className="text-lg font-bold text-blue-600 underline mt-2">  
    Tailwind makes styling easy!  
</p>
```

- ✓ **text-lg** - Sets the font size to large (lg), which is typically 18px (1.125rem).
- ✓ **font-bold** - Makes the text bold by applying **font-weight: 700**.
- ✓ **text-blue-600** - Sets the text color to a shade of blue (600 is a darker shade).
- ✓ **underline** - Adds an **underline** to the text.
- ✓ **mt-2** - Adds a top margin (mt) of 2, which corresponds to 0.5rem (8px).

Spacing (Margin & Padding) example

```
<div class="p-4 m-4 bg-gray-200">  
    This box has padding and margin.  
</div>
```

- ✓ **p-4** → Adds padding (p) of 4, which corresponds to 1rem (16px) on all sides (top, bottom, left, and right).
- ✓ **m-4** → Adds margin (m) of 4, which corresponds to 1rem (16px) on all sides.
- ✓ **bg-gray-200** → Sets the background color to a light gray shade (gray-200).

Using Tailwind CSS in React

eazy
bytes

Background & Border

```
<div className="bg-green-300 border border-green-600 p-4 rounded-lg">  
  Green box with border and rounded corners  
</div>
```

- ✓ `bg-green-300` → Sets the background color to a light green shade (green-300).
- ✓ `border` → Adds a default border (1px solid).
- ✓ `border-green-600` → Changes the border color to a darker green shade (green-600).
- ✓ `p-4` → Adds padding of 4 units (1rem or 16px) on all sides.
- ✓ `rounded-lg` → Adds rounded corners with a large radius (lg).

Flexbox (Alignment & Layout) example

```
<div className="flex justify-center items-center gap-4 bg-gray-300 mt-4 flex-col">  
  <p className="text-xl font-bold">Text 1</p>  
  <p className="text-xl font-bold">Text 2</p>  
  <p className="text-xl font-bold">Text 3</p>  
</div>
```

- ✓ `flex` → Enables Flexbox layout.
- ✓ `justify-center` → Centers the elements horizontally.
- ✓ `items-center` → Centers the elements vertically.
- ✓ `gap-4` → Adds spacing of 16px (1rem) between child elements.
- ✓ `bg-gray-300` → Applies a light gray background.
- ✓ `mt-4` → Adds a top margin of 16px (1rem).
- ✓ `flex-col` → Displays the items in column(vertical) style

Using Tailwind CSS in React

eazy
bytes

Grid Layout example

```
<div className="grid grid-cols-3 gap-4 mt-4">  
  <div className="bg-blue-200 p-4 text-red-700">1</div>  
  <div className="bg-blue-400 p-4">2</div>  
  <div className="bg-blue-600 p-4 text-white">3</div>  
</div>
```

- ✓ `grid` → Enables CSS Grid layout.
- ✓ `grid-cols-3` → Creates 3 equal columns.
- ✓ `gap-4` → Adds a 16px (1rem) gap between grid items.
- ✓ `mt-4` → Adds a top margin of 16px (1rem).

Button Styling example

```
<button className="px-6 py-2 m-4 bg-blue-500 text-white rounded-lg hover:bg-blue-700  
  transition duration-300">Click Me</button>
```

- ✓ `px-6` → Horizontal padding of 24px (1.5rem).
- ✓ `py-2` → Vertical padding of 8px (0.5rem).
- ✓ `m-4` → Margin of 16px (1rem) around the button.
- ✓ `bg-blue-500` → Blue background (#3B82F6).
- ✓ `text-white` → White text.
- ✓ `rounded-lg` → Large rounded corners.
- ✓ `hover:bg-blue-700` → On hover, background changes to a darker blue (#1D4ED8).
- ✓ `transition duration-300` → Smooth transition effect over 300ms.

Using Tailwind CSS in React

eazy
bytes

Animations & Effects example

```
<button className="px-6 py-2 m-4 bg-red-500 text-white rounded-lg hover:bg-red-700 transition duration-500 ease-in-out transform hover:scale-110">Hover Me</button>
```

- ✓ `hover:bg-red-700` → On hover, background changes to a darker red (#B91C1C).
- ✓ `transition duration-500` → Smooth transition effect over 500ms.
- ✓ `ease-in-out` → Eases both in and out for a smooth effect.
- ✓ `transform hover:scale-110` → On hover, the button scales up (110%), creating a zoom-in effect.

Card Component example

```
<div className="max-w-sm rounded-lg shadow-lg bg-indigo-100 p-6">
  <h2 className="text-2xl font-bold mb-2">Tailwind Card</h2>
  <p className="text-gray-700">
    This is a simple card built with Tailwind CSS.
  </p>
  <button className="mt-4 px-6 py-2 bg-black text-white rounded-lg hover:bg-blue-700 transition">
    Read More
  </button>
</div>
```

- ✓ `max-w-sm` → Limits the card width to small (sm), ensuring responsiveness.
- ✓ `shadow-lg` → Adds a large shadow, creating depth.
- ✓ `transition` → Ensures smooth hover effect.

Using Tailwind CSS in React

eazy
bytes

Responsive Design example

```
<p className="text-red-600 sm:text-blue-600 md:text-emerald-600 lg:text-yellow-500  
xl:text-purple-600">  
  Resize the screen to see changes!  
</p>
```

- ✓ `text-red-600` → Default text color is red (#DC2626).
- ✓ `sm:text-blue-600` → On small screens ($\geq 640\text{px}$), text color changes to blue (#2563EB).
- ✓ `md:text-emerald-600` → On medium screens ($\geq 768\text{px}$), text color changes to emerald (#059669).
- ✓ `lg:text-yellow-500` → On large screens ($\geq 1024\text{px}$), text color changes to yellow (#EAB308).
- ✓ `xl:text-purple-600` → On extra-large screens ($\geq 1280\text{px}$), text color changes to purple (#9333EA).

Adding custom styles in Tailwind

Tailwind is a utility-first framework, but sometimes, you need custom styles. Using below syntax, we can implement custom styles,

```
<div className="w-[500px] h-[200px] bg-[#ff5733]">  
    Custom Width & Color  
</div>
```

- ✓ w-[500px] → Sets width to 500px.
- ✓ h-[200px] → Sets height to 200px.
- ✓ bg-[#ff5733] → Custom HEX color.

Theme variables

In Tailwind CSS v4.0, the framework introduces a more streamlined approach to theme customization by leveraging CSS variables through the `@theme` directive. This method enhances flexibility and simplifies the process of defining design tokens directly within your CSS files.

For example, the following setup defines a primary color and a display font that can be utilized throughout your project.

```
@import "tailwindcss";  
  
@theme {  
  --color-primary: #aab9ff;  
  --font-display: "Poppins", sans-serif;  
}
```

Defining theme variables automatically generates corresponding utility classes, enabling you to apply these styles directly in your HTML.

Example:

```
<div className="bg-primary font-display"> Tailwind CSS v4.0 makes theming easier! </div>
```

Here, `bg-primary` and `font-display` are utilities generated from the theme variables defined earlier.

Key Differences Between Tailwind and Bootstrap

Feature	Tailwind CSS	Bootstrap
Approach	Utility-first	Component-based
Customization	Highly customizable	Customization via overriding
Design system	No predefined UI components	Comes with prebuilt UI components
Learning Curve	Medium	Easy (familiar UI components)
Performance	Smaller CSS size (JIT mode)	Includes unused styles (larger file size)
Example Usage	<code><h1 class="text-2xl font-bold">Welcome</h1></code>	<code><button class="btn btn-primary" type="submit">Button</button></code>

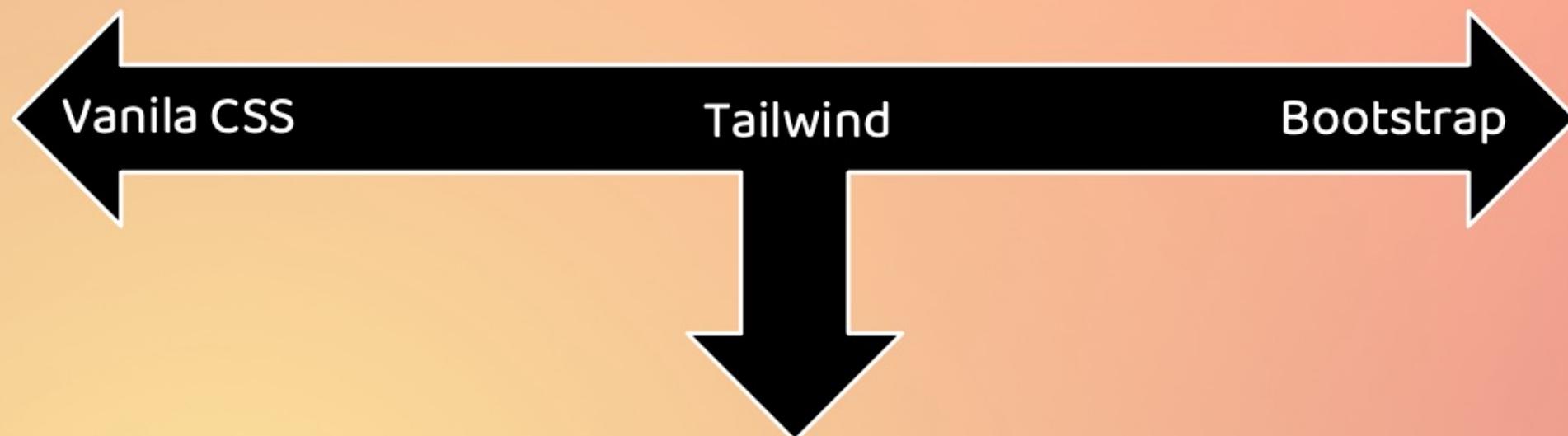
CSS, Tailwind and Bootstrap

In the world of frontend styling, we have two extremes:

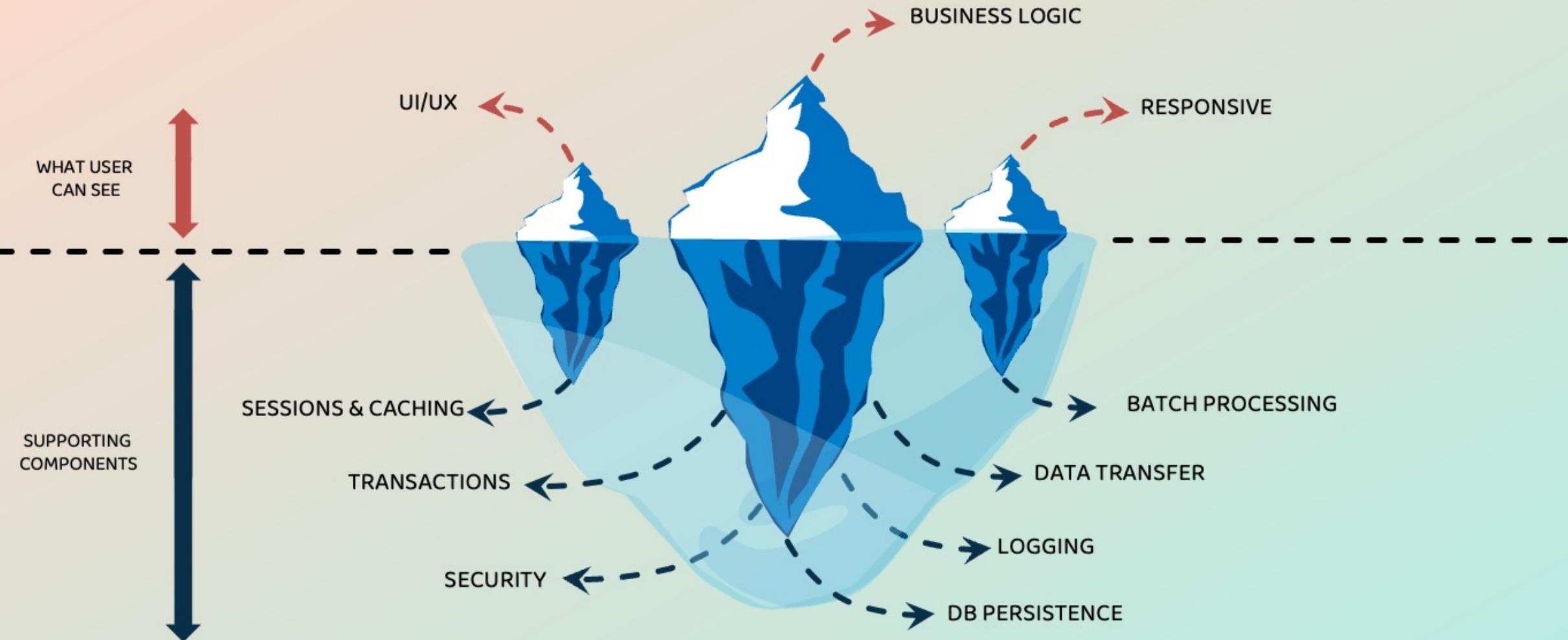
Vanilla CSS (Extreme Left) → Writing raw CSS from scratch gives full control but requires a lot of effort for structuring, maintaining, and making designs responsive.

Bootstrap (Extreme Right) → Provides pre-styled components like buttons, forms, and modals, making it quick to build UI. However, it limits customization and often leads to a "Bootstrap look" in projects.

Tailwind bridges the gap between manually writing CSS and using prebuilt frameworks. It combines customization, flexibility, and speed, making it the best choice for modern UI development.



Behind the scenes of a web app



Why should we use frameworks ?



CHEF VICKY

Uses best readily available best ingredients like Cheese, Pizza Dough etc. to prepare Pizza



Pizza preparation time is less



Can easily scale his restaurant pizza orders



Gets consistent taste for his pizzas



Focus more on the pizza preparation



Less efforts and more results/revenue



CHEF SANJEEV

Prepare all the ingredients like Cheese, Pizza Dough etc. by himself to prepare Pizza



Pizza preparation time is more



Scaling his restaurant pizza orders is not an option



May not get a consistent taste for his pizzas



Focus more on the raw material & ingredients



More efforts and less results/revenue

Why should we use frameworks ?



DEV SANJEEV

Uses best readily available best frameworks like Spring, Spring Boot etc. to build a web app



Leverage Security, Logging etc. from frameworks



Can easily scale his application



App will work in an predictable manner



Focus more on the business logic



Less efforts and more results/revenue



DEV VICKY

Build his own code by himself to build a web app



Need to build code for Security, Logging etc.



Scaling his is not an option till he test everything



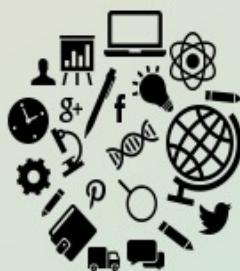
App may not work in an predictable manner



Focus more on the supporting components



More efforts and less results/revenue



What is Spring ?

Spring is a popular Java framework for building web applications.

It helps developers write clean, modular, and testable code.

Provides built-in features for handling databases, security, messaging, and more.

Why Spring ?

Makes Java development easier. Helps in reducing boilerplate code.

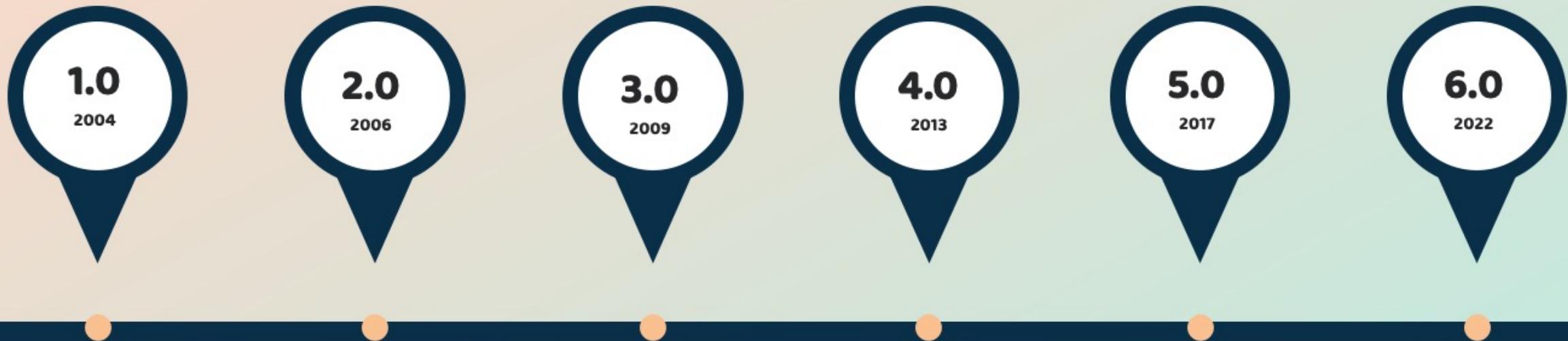
Supports different types of applications – web apps, microservices, cloud apps, etc.

Provides dependency injection (helps in managing object dependencies).

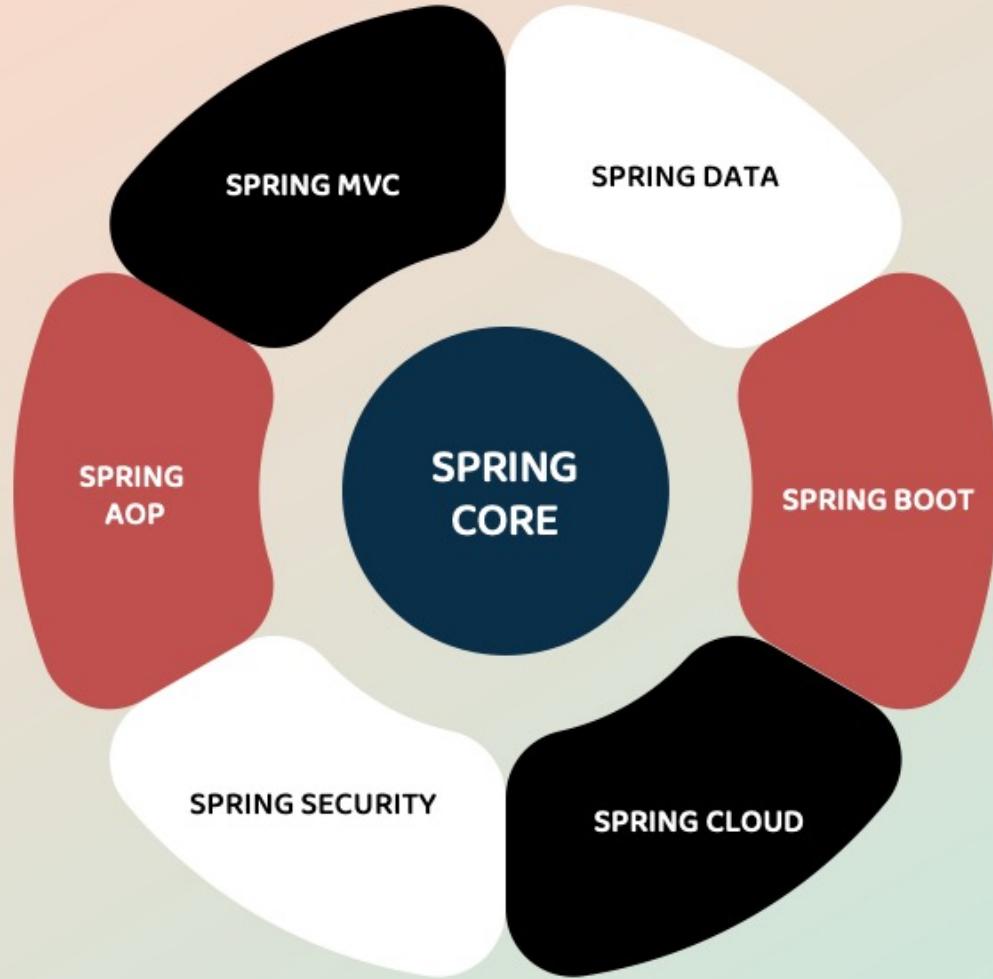
Works well with other technologies (like Hibernate, Kafka, RabbitMQ)

Introduced in 2003, the Spring Framework is a comprehensive solution designed to simplify Java enterprise development. It was created to tackle the challenges of building large-scale applications by offering a unified programming model and seamless integration with various technologies.

Spring Release timeline



- The first version of Spring was created by Rod Johnson, introduced with his book *Expert One-on-One J2EE Design and Development* in October 2002.
- Spring emerged in 2003 to simplify the complexities of early J2EE specifications. Instead of competing with Java EE, Spring complements it by integrating only selected features from the Java EE ecosystem.
- Over the years, Spring has continued to grow and evolve. Beyond the main framework, other popular projects like Spring Boot, Spring Security, Spring Data, Spring Cloud, and Spring Batch have been introduced.



What is Spring Core ?

Spring Core is the foundation of the entire Spring ecosystem. It provides the fundamental principles, base classes, and core mechanisms that power all other Spring modules and projects.

The entire Spring Framework and its related projects are built on top of Spring Core.

Key Components of Spring Core

Spring Core consists of the following essential components:

IoC (Inversion of Control) – Shifts control of object creation to the Spring Container.

DI (Dependency Injection) – Manages dependencies automatically, reducing tight coupling.

Beans – Objects managed by the Spring IoC container.

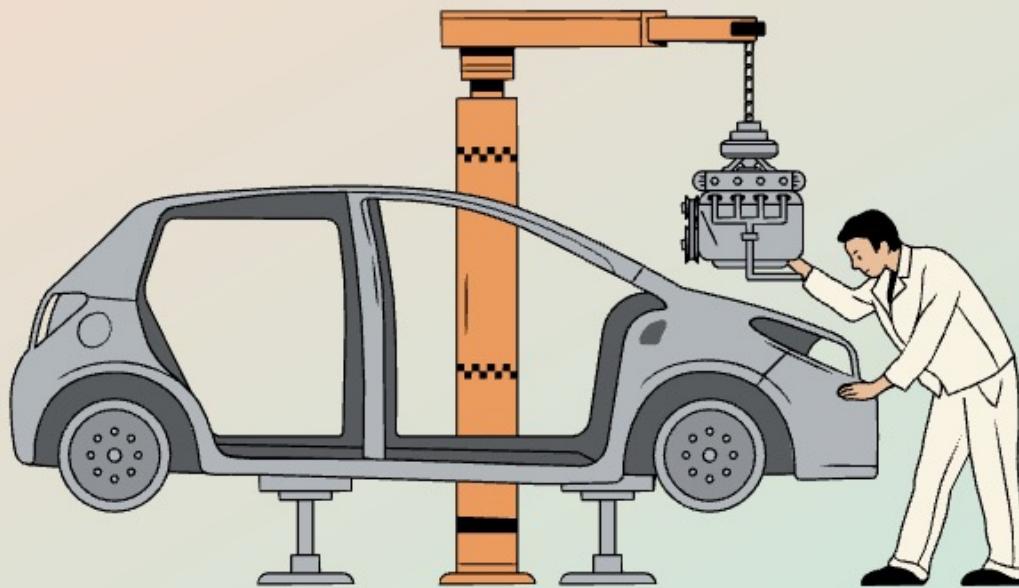
Context – Provides runtime configuration and manages the lifecycle of beans.

IoC Container – The core mechanism that creates, manages, and provides dependencies for beans

Spring Framework is a powerful **foundation** for enterprise applications.

Spring Boot simplifies the development by **automating configurations, embedding servers, and providing ready-to-use features**.

Spring Boot is **NOT a replacement for Spring**, but rather an extension that makes Spring easier to use.



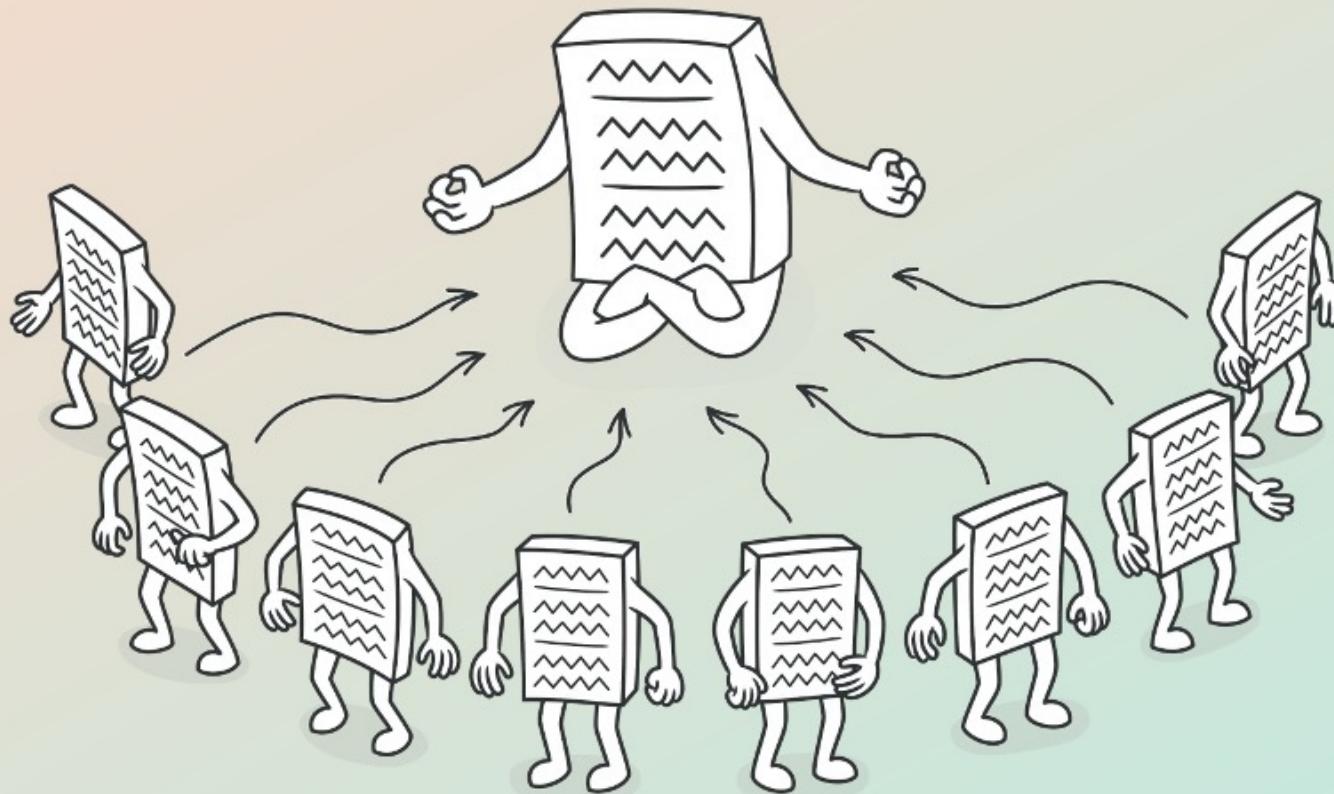
Think of **Spring** as a **car engine** and **Spring Boot** as a **Fully assembled car**.

Spring Framework (Engine): Provides the core functionality needed to build Java applications, but requires manual configurations (like adding dependencies, configuring servers, etc.).

Spring Boot (Assembled Car): Makes development faster by **automating configurations**, embedding a web server, and providing default settings, so developers can focus on writing business logic instead of managing configurations.

Inversion of Control (IoC) is a software design principle that defines how objects are created and managed in a program. It doesn't create the objects itself but outlines a way for their creation and dependency management.

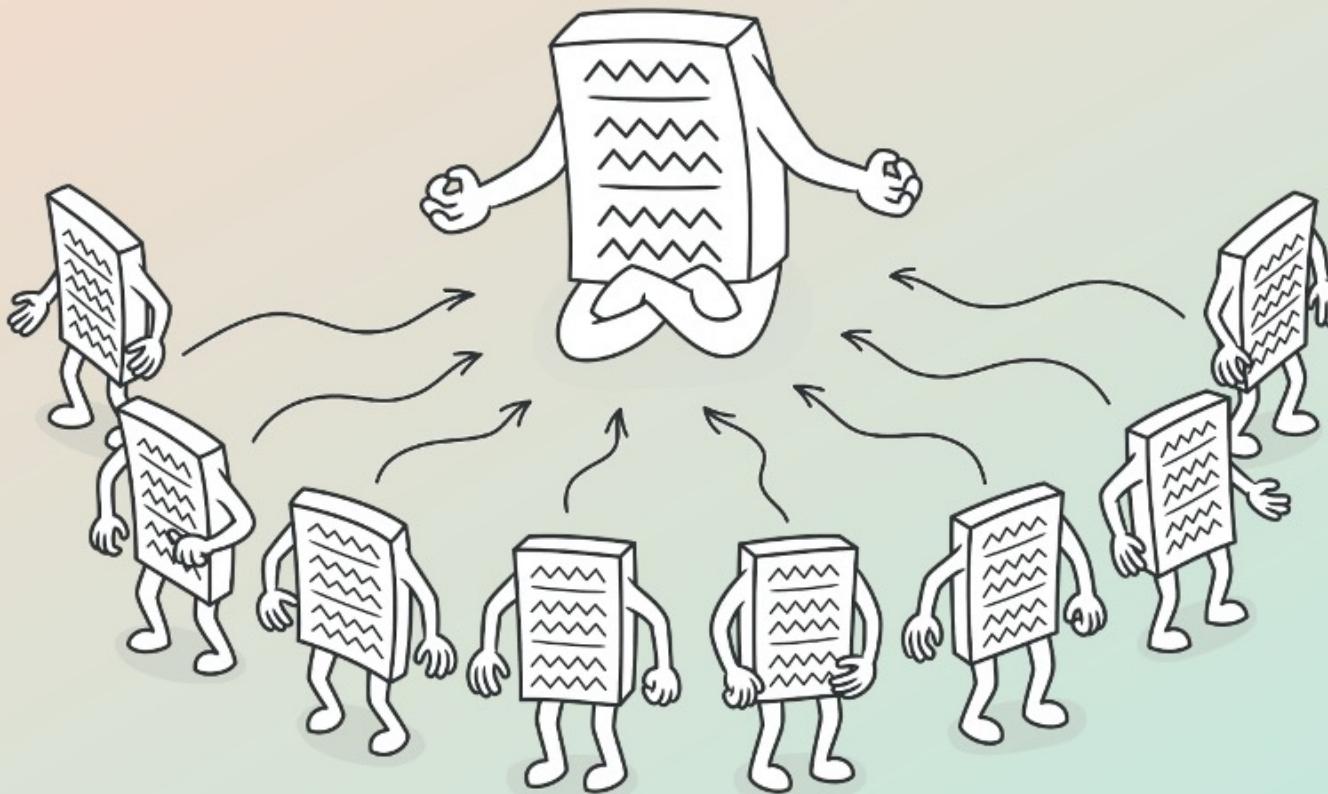
With IoC, the control flow of the program is reversed. Instead of the programmer managing the flow of the application, a framework or service takes over this responsibility, ensuring objects and their dependencies are handled automatically.



- ✓ Normally, in Java, we create objects manually using new keyword.
- ✓ With IoC, Spring manages object creation and dependencies automatically.
- ✓ It reduces manual coding and makes applications more flexible.

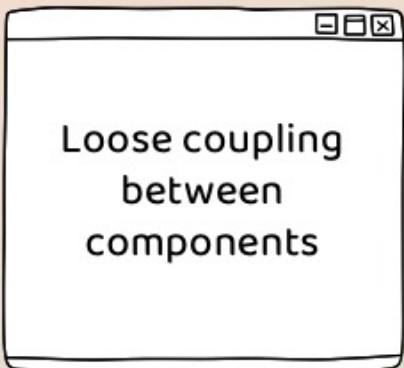
Dependency Injection is a design pattern used to implement Inversion of Control (IoC).

With Dependency Injection, the responsibility of creating and managing objects is transferred from the application to the Spring IoC container. This approach reduces coupling between objects by allowing the framework to dynamically provide the required dependencies at runtime.

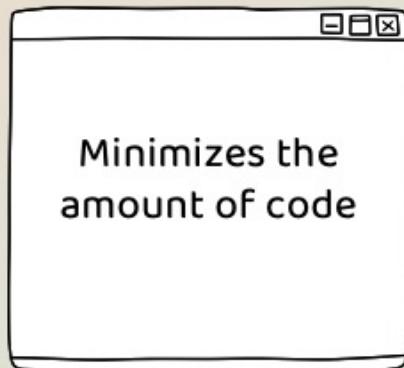


- ✓ DI is a way of providing dependencies to a class instead of creating them inside the class.
- ✓ Helps in bringing loose coupling between objects
- ✓ Spring automatically injects required objects.

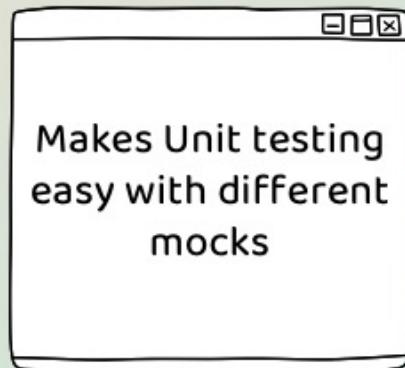
Advantages of IoC & DI



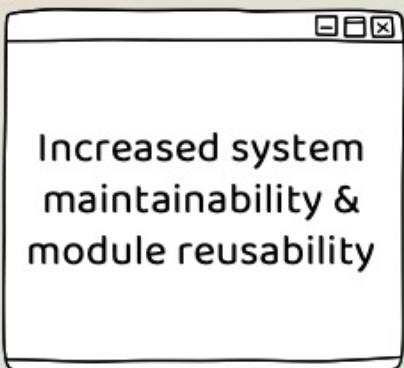
Loose coupling
between
components



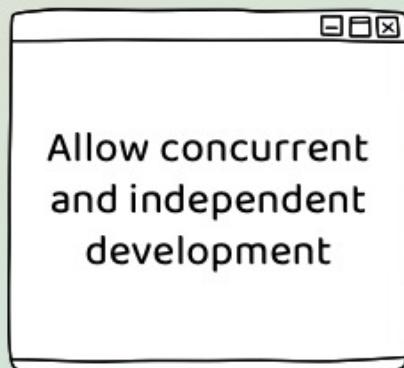
Minimizes the
amount of code



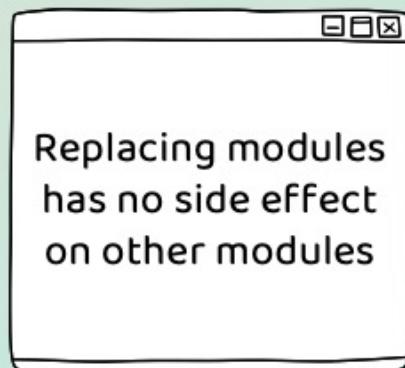
Makes Unit testing
easy with different
mocks



Increased system
maintainability &
module reusability



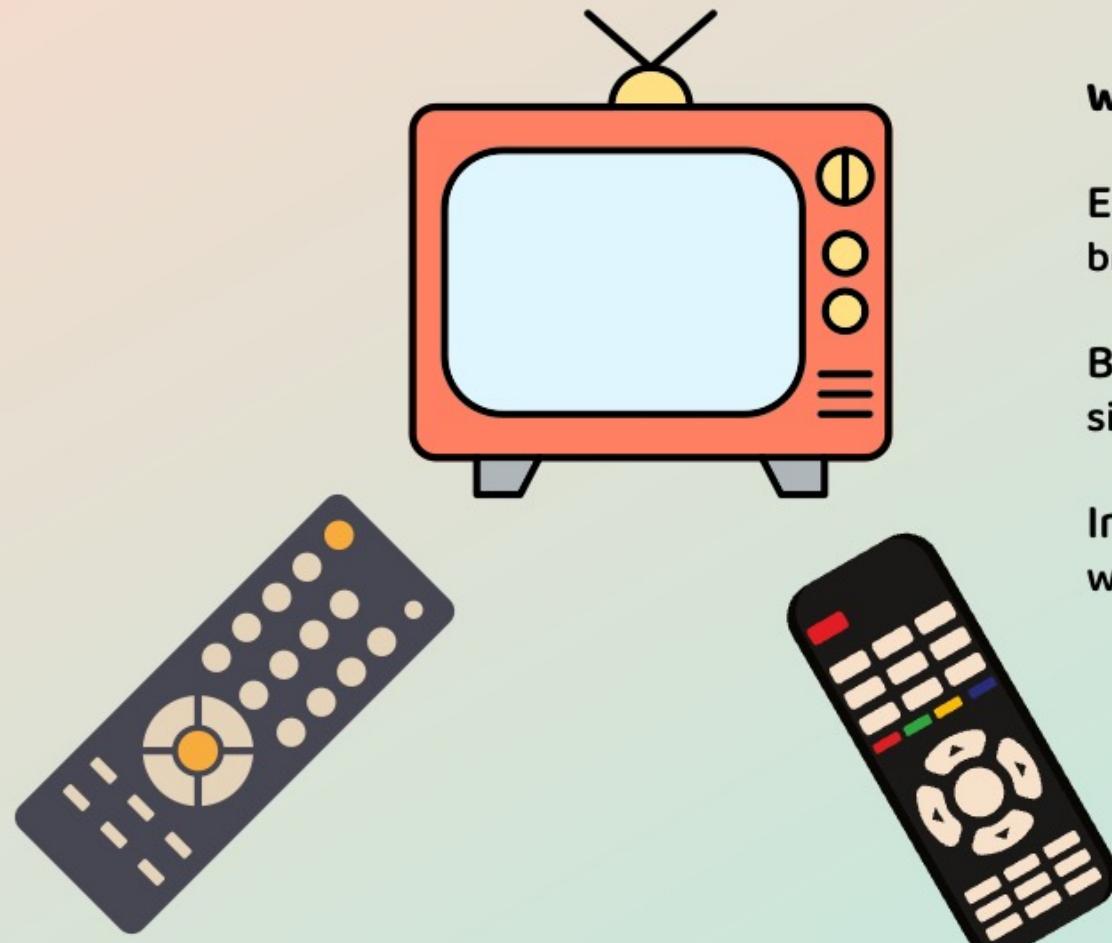
Allow concurrent
and independent
development



Replacing modules
has no side effect
on other modules

Loose coupling means that different parts of a system are not tightly dependent on each other. Instead, they are connected in a way that changes in one part have little or no impact on others.

Imagine using a TV remote. The remote can control different TVs, and the TV can work with different remotes. Neither depends on the specific details of the other—they just need to follow a common way of communicating (like the infrared signals).



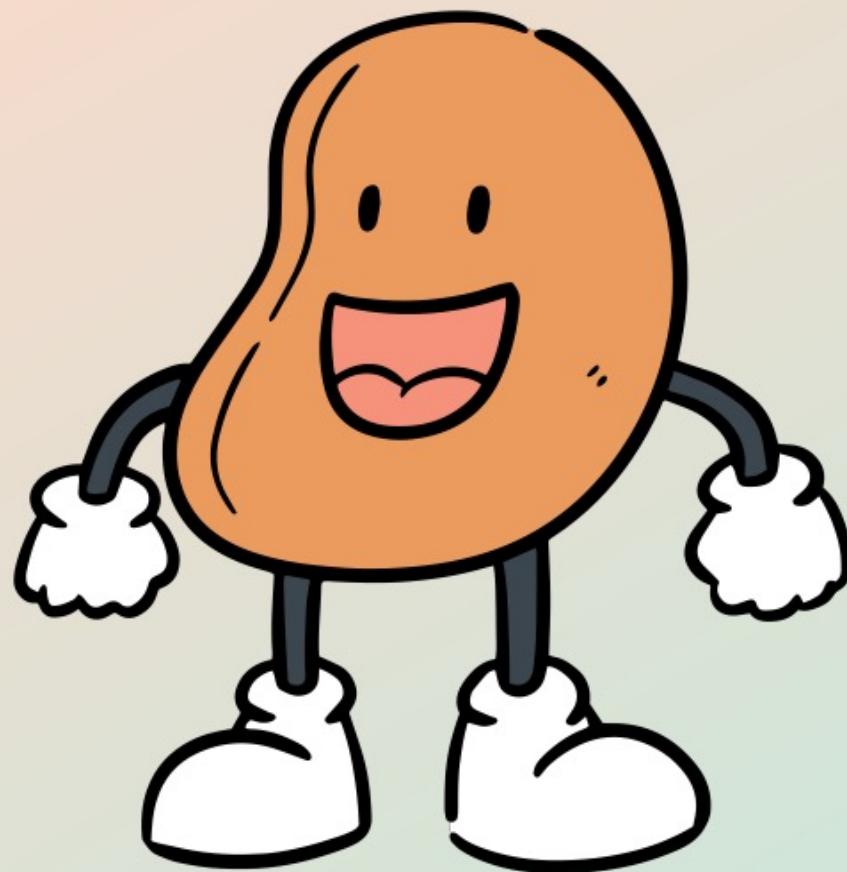
Why Loose Coupling Is Important

Easier to Modify: You can change one part of the system without breaking others.

Better Reusability: Components can be reused in different contexts since they don't depend on specific implementations.

Improved Flexibility: You can replace or upgrade parts of the system without affecting others.

A Spring Bean is any normal Java class that is instantiated and managed by the Spring IoC (Inversion of Control) container. These beans form the backbone of a Spring application, representing its key components or services.



How Beans Are Created:

Beans are instantiated by the container based on the configuration metadata developer provide, which can be using:

- XML configurations (Legacy approach)
- Annotations (e.g., @Component, @Bean, @Service)
- Java-based configuration (via @Configuration and @Bean methods)

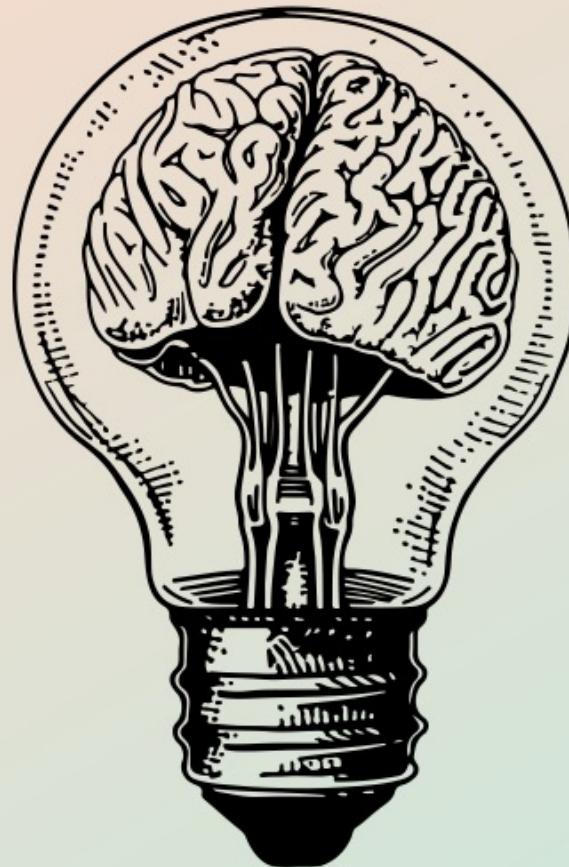
Bean Lifecycle Management:

The Spring IoC container manages the entire lifecycle of a bean, including its creation, initialization, scope, and destruction.

It also ensures that any required dependencies are injected into the bean as needed.

The Spring Context is like the "memory" of your application where Spring manages all the objects (beans) it needs to handle.

By default, Spring doesn't know about the objects or components you define in your application. To make Spring aware of them, you need to register these objects into the context.



To make Spring aware of your objects, you need to register them with the context using:

- Annotations like `@Component`, `@Service`, or `@Repository`.
- Explicit configurations in XML or Java.

The IoC (Inversion of Control) Container is the core of the Spring Framework. Its main job is to manage the objects (also known as Spring Beans) in your application. It is responsible for:

- Creating instances (Beans) of your application classes.
- Configuring objects (e.g., setting their properties or initializing dependencies).
- Managing dependencies between objects using Dependency Injection (DI).



Types of IoC Containers in Spring

There are two main types of IoC containers in Spring:

`org.springframework.beans.factory.BeanFactory`:

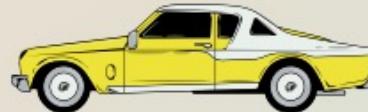
- ✓ The simplest IoC container in Spring.
- ✓ It provides the basic functionality for managing Spring Beans.
- ✓ Suitable for lightweight applications where performance is critical and fewer features are needed.

`org.springframework.context.ApplicationContext`:

- ✓ A more advanced IoC container, built on top of BeanFactory.
- ✓ It adds extra features like: Internationalization (i18n) support, Event propagation and listeners etc

How to add Beans into Spring Context ?

When we create an java object with new () operator directly as shown below, then your Spring Context/Spring IoC Container will not have any clue of the object.



```
Vehicle veh = new Vehicle();
```

Spring Context



@Bean annotation lets Spring know that it needs to call this method when it initializes its context and adds the returned object/value to the Spring context/Spring IoC Container.

```
@Bean  
Vehicle vehicle() {  
    var veh = new Vehicle();  
    veh.setName("Tesla");  
    return veh;  
}
```

Spring Context



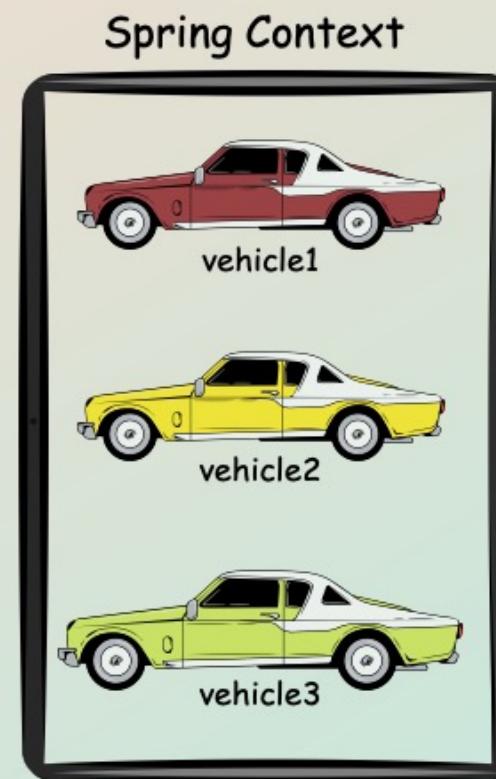
What is NoUniqueBeanDefinitionException ?

When multiple beans of the same type are defined in the Spring context, and we attempt to retrieve a bean by type, Spring cannot determine which instance to provide. This ambiguity results in a `NoUniqueBeanDefinitionException`, as shown below.

```
@Bean
Vehicle vehicle1() {
    var veh = new Vehicle();
    veh.setName("Audi");
    return veh;
}

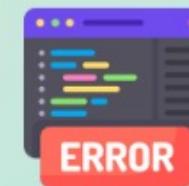
@Bean
Vehicle vehicle2() {
    var veh = new Vehicle();
    veh.setName("Honda");
    return veh;
}

@Bean
Vehicle vehicle3() {
    var veh = new Vehicle();
    veh.setName("Ferrari");
    return veh;
}
```



```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Vehicle veh = context.getBean(Vehicle.class);
```

NoUniqueBeanDefinitionException



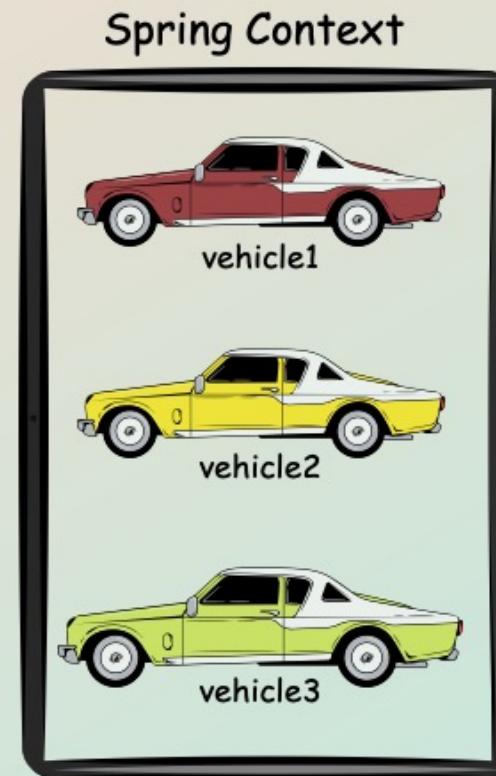
How to solve NoUniqueBeanDefinitionException ?

To avoid NoUniqueBeanDefinitionException in these kind of scenarios, we can fetch the bean from the context by mentioning it's name like shown below,

```
@Bean
Vehicle vehicle1() {
    var veh = new Vehicle();
    veh.setName("Audi");
    return veh;
}

@Bean
Vehicle vehicle2() {
    var veh = new Vehicle();
    veh.setName("Honda");
    return veh;
}

@Bean
Vehicle vehicle3() {
    var veh = new Vehicle();
    veh.setName("Ferrari");
    return veh;
}
```



```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Vehicle veh = context.getBean("vehicle1",Vehicle.class);
```



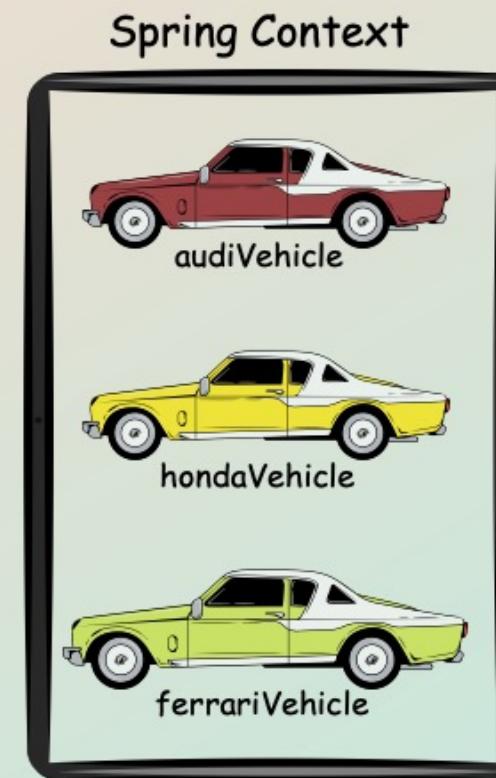
Different ways to name a Bean

By default, Spring will consider the method name as the bean name. But if we have a custom requirement to define a separate bean name, then we can use any of the below approach with the help of @Bean annotation,

```
@Bean(name="audiVehicle")
Vehicle vehicle1() {
    var veh = new Vehicle();
    veh.setName("Audi");
    return veh;
}

@Bean(value="hondaVehicle")
Vehicle vehicle2() {
    var veh = new Vehicle();
    veh.setName("Honda");
    return veh;
}

@Bean("ferrariVehicle")
Vehicle vehicle3() {
    var veh = new Vehicle();
    veh.setName("Ferrari");
    return veh;
}
```



```
Vehicle veh1 = context.getBean("audiVehicle",Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh1.getName());

Vehicle veh2 = context.getBean("hondaVehicle",Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh2.getName());

Vehicle veh3 = context.getBean("ferrariVehicle",Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh3.getName());
```

Output on console:

```
Vehicle name from Spring Context is: Audi
Vehicle name from Spring Context is: Honda
Vehicle name from Spring Context is: Ferrari
```

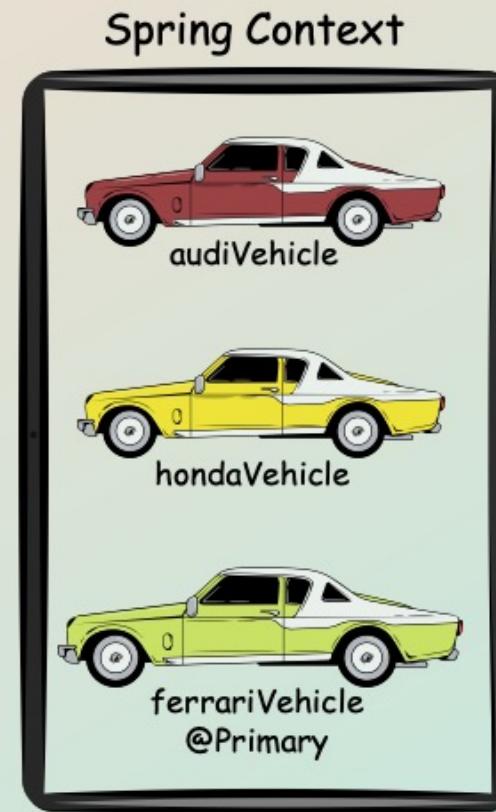
When multiple beans of the same type exist in the Spring context, you can designate one as the default by using the `@Primary` annotation.

The primary bean is the one Spring will automatically select when multiple candidates are available, and no specific bean name is provided. In other words, `@Primary` helps resolve ambiguity by instructing Spring which bean to prioritize when multiple options exist.

```
@Bean(name="audiVehicle")
Vehicle vehicle1() {
    var veh = new Vehicle();
    veh.setName("Audi");
    return veh;
}

@Bean(value="hondaVehicle")
Vehicle vehicle2() {
    var veh = new Vehicle();
    veh.setName("Honda");
    return veh;
}

@Primary
@Bean("ferrariVehicle")
Vehicle vehicle3() {
    var veh = new Vehicle();
    veh.setName("Ferrari");
    return veh;
}
```

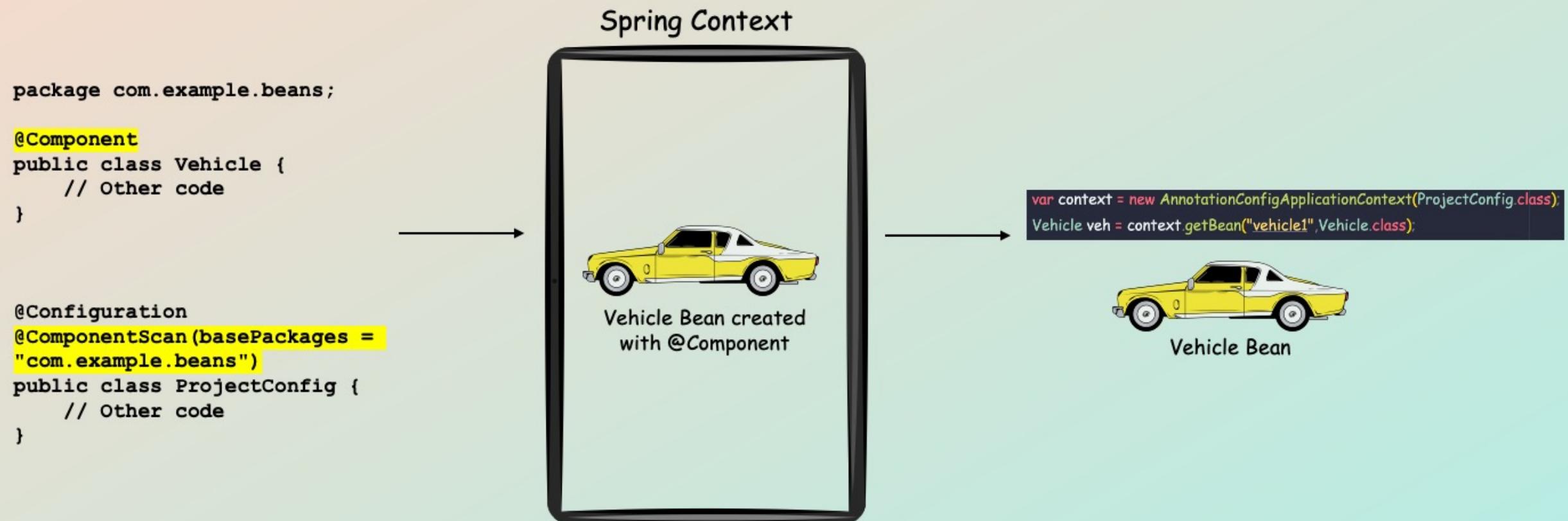


```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Vehicle vehicle = context.getBean(Vehicle.class);
```



@Component is one of the most commonly used stereotype annotations in Spring. It simplifies the process of creating and registering a bean in the Spring context with minimal code, compared to the @Bean method. By applying @Component to a class, Spring automatically detects and manages it as a Spring Bean.

To ensure Spring scans and registers these annotated classes, you can use the @ComponentScan annotation over a configuration class.



@PostConstruct is a Spring annotation used to run a method automatically after a Spring bean is created and its dependencies are injected.

When is @PostConstruct used?

- To Perform setup logic when a bean is created.
- To Load data, set default values, or execute startup logic.
- To Initialize resources after dependency injection.

Example

```
@Component
public class Vehicle {

    private String name;

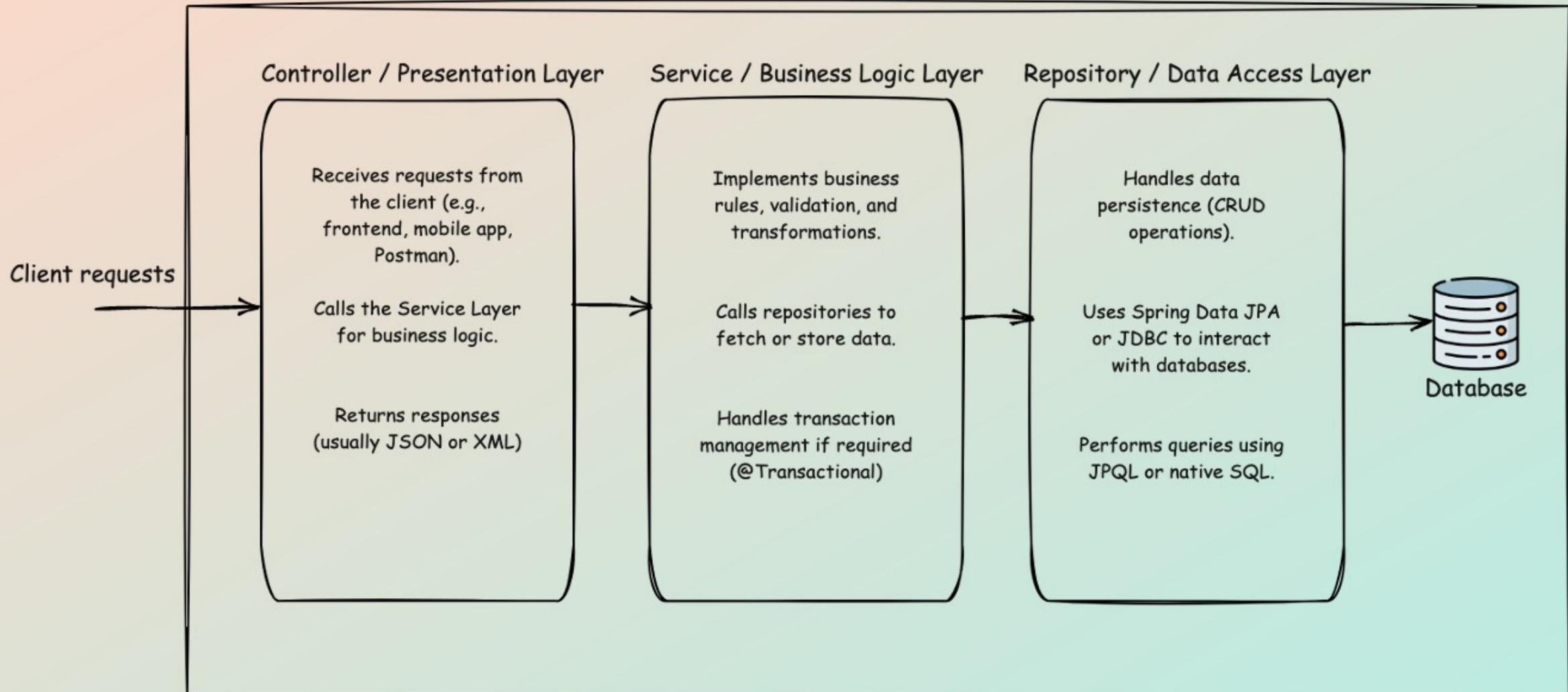
    @PostConstruct
    public void initialize() {
        this.name = "Honda";
    }
}
```

When Does @PostConstruct Run?

- It runs after the constructor and after all dependencies are injected.
- It runs only once in the bean's lifecycle.

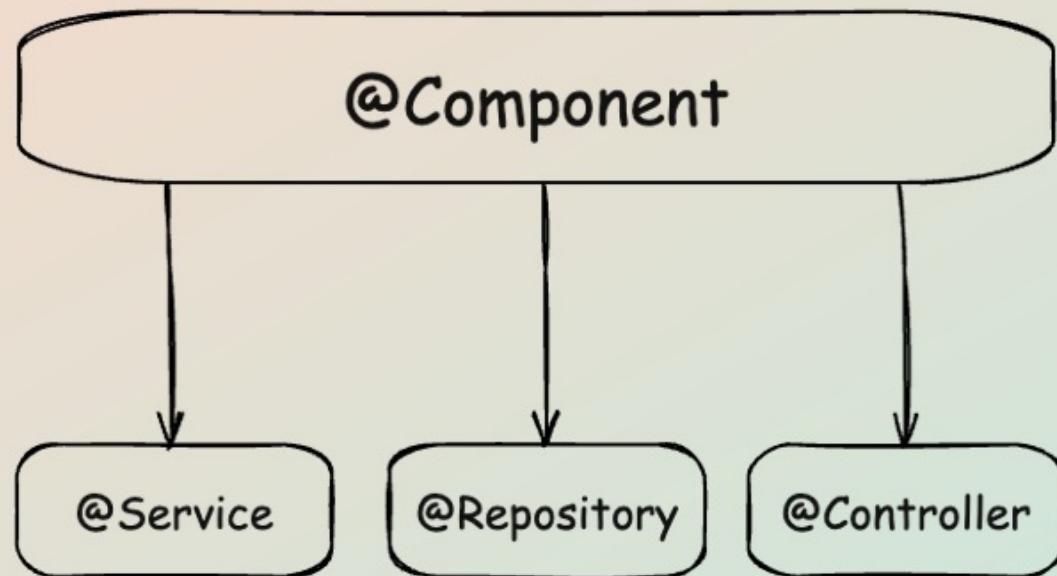
- Spring creates Vehicle bean when the application starts.
- Before using the bean, it calls the initialize() method automatically.
- This is useful for setting up initial data or configurations.

Layers in a Backend application



Spring provides stereotype annotations, which are special annotations that automatically register Spring beans in the application context. These annotations help simplify bean creation and management.

The key stereotype annotations in Spring include: `@Component`, `@Service`, `@Repository`, `@Controller`



`@Component` → Generic bean (used for any class).

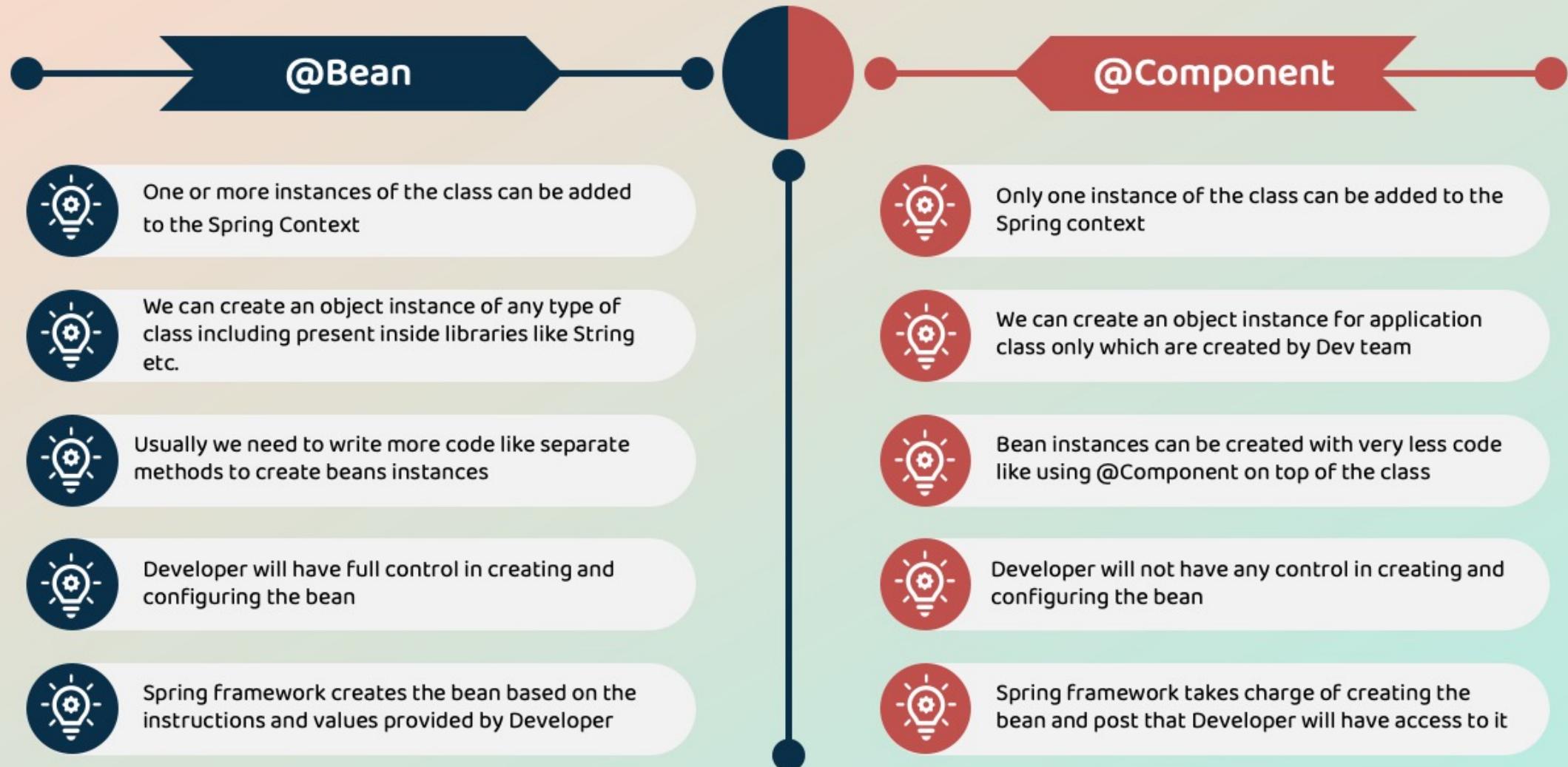
`@Service` → Business logic layer.

`@Repository` → Data access layer

`@Controller` → Handles incoming web requests in Spring MVC

`@RestController` → Simplified `@Controller`, returns JSON responses.
Used to build REST APIs

Use `@ComponentScan` to enable automatic scanning of these annotations.



Introduction to Beans Wiring inside Spring

In Java web applications, objects often delegate tasks to other objects, creating dependencies between them. Similarly, in Spring, when we define multiple beans, some beans may rely on others to function correctly. **Bean wiring** is the process of connecting these dependent beans within the Spring IoC container.

Manually managing these dependencies can become complex, so Spring provides automatic wiring (**Autowiring**) to simplify the process.

Spring Context
with out wiring



Spring Context
with wiring & DI



No wiring scenario inside spring

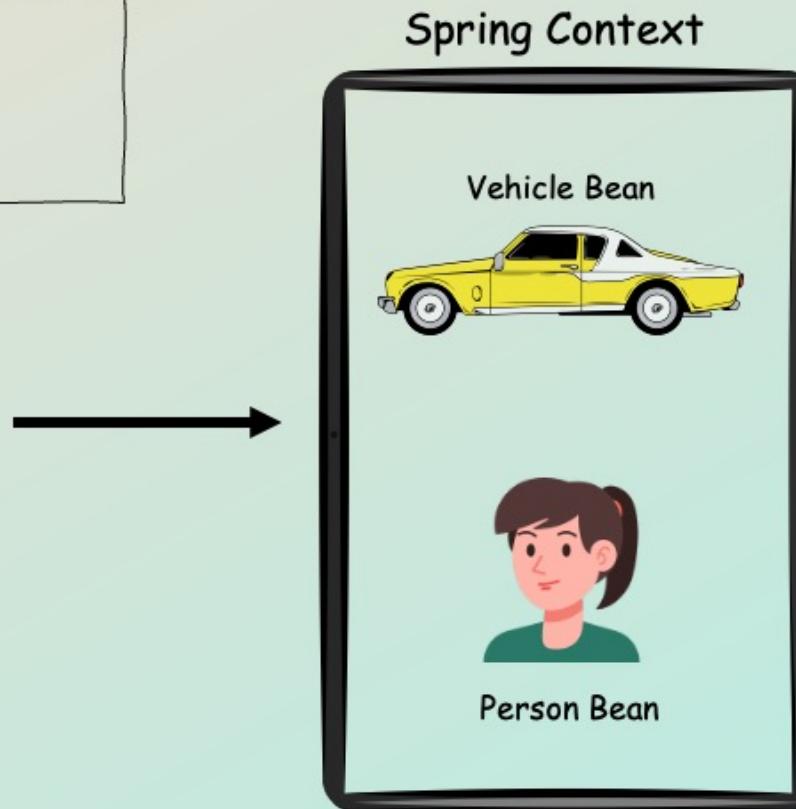
Consider a scenario where we have two java classes Person and Vehicle. The Person class has a dependency on the Vehicle. Based on the below code,

we are only creating the beans inside the Spring Context and no wiring will be done. Due to this both this beans present inside the Spring context with out knowing about each other.

```
public class Person {  
  
    private String name;  
    private Vehicle vehicle;  
  
}
```

```
public class Vehicle {  
  
    private String name;  
  
}
```

```
@Bean  
public Vehicle vehicle() {  
    Vehicle vehicle = new Vehicle();  
    vehicle.setName("Toyota");  
    return vehicle;  
}  
  
  
@Bean  
public Person person() {  
    Person person = new Person();  
    person.setName("Lucy");  
    return person;  
}
```

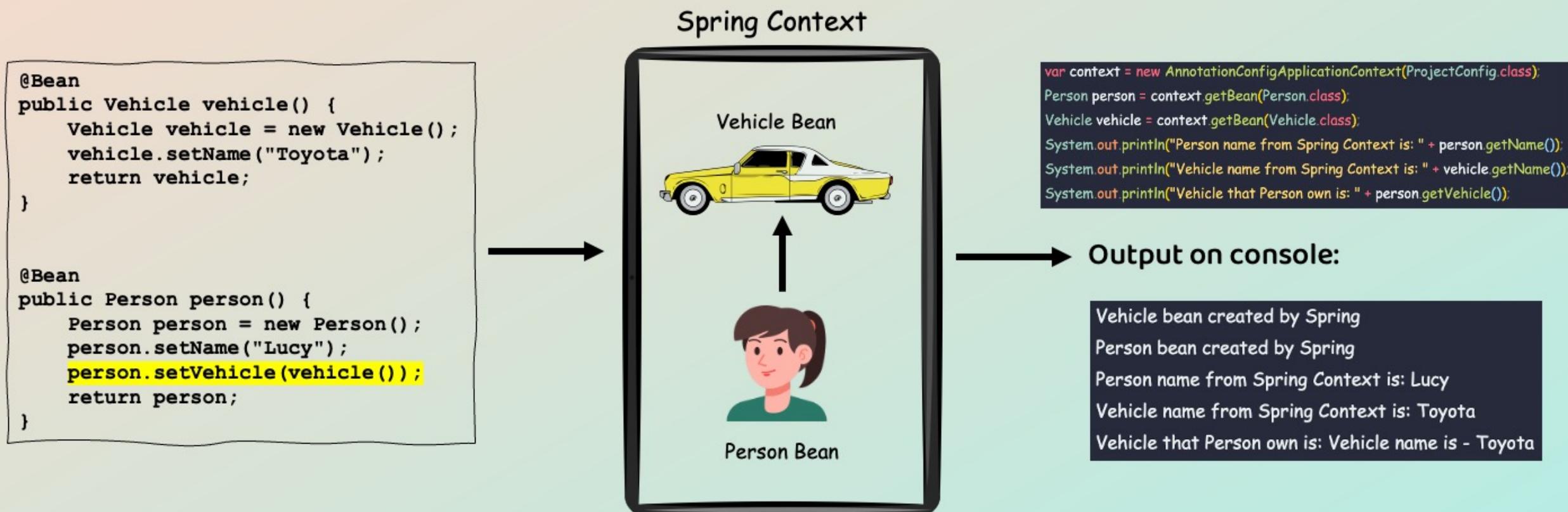


Vehicle doesn't belong to any person. The Person and Vehicle beans present in context but no relation established

Wiring Beans using method call

In the code below, we establish a relationship between the Person and Vehicle beans by calling the vehicle() bean method from within the person() bean method. This ensures that the Person bean owns a Vehicle within the Spring context.

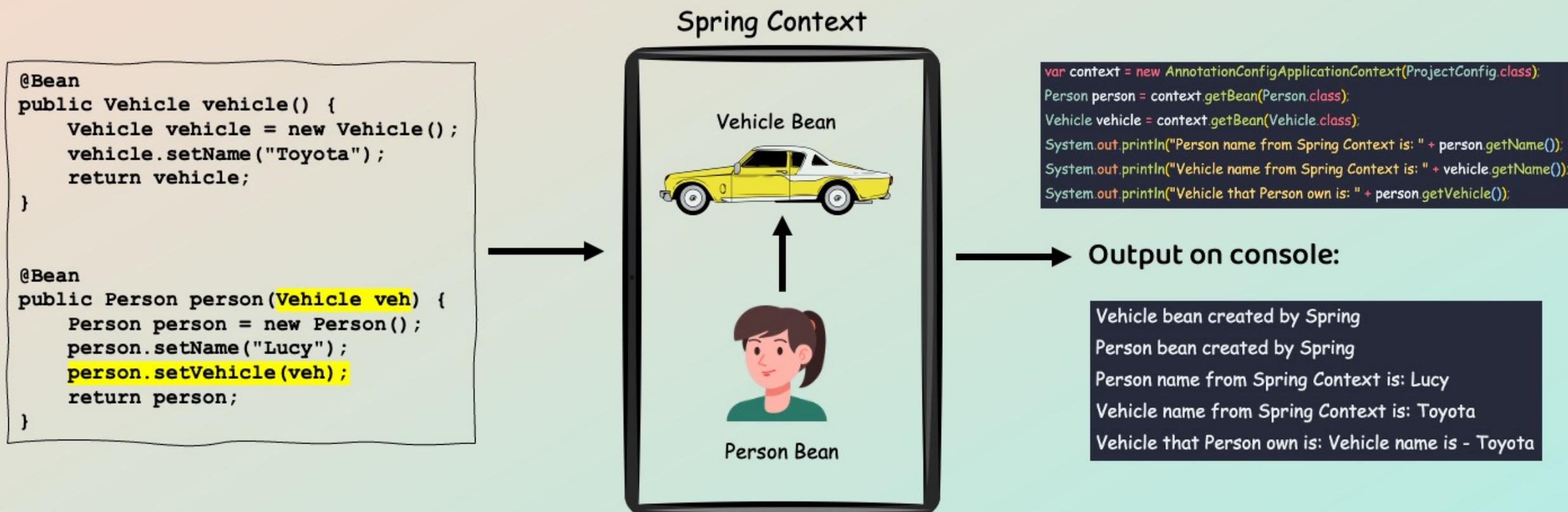
Spring automatically ensures that only one instance of the Vehicle bean is created and manages its lifecycle efficiently. Since the Person bean depends on the Vehicle bean, Spring will always instantiate the Vehicle bean first before creating the Person bean.



Wiring Beans using method parameters

In the code below, we establish a relationship between the Person and Vehicle beans by passing the Vehicle bean as a method parameter to the person() bean method. This ensures that within the Spring context, the Person bean owns a Vehicle.

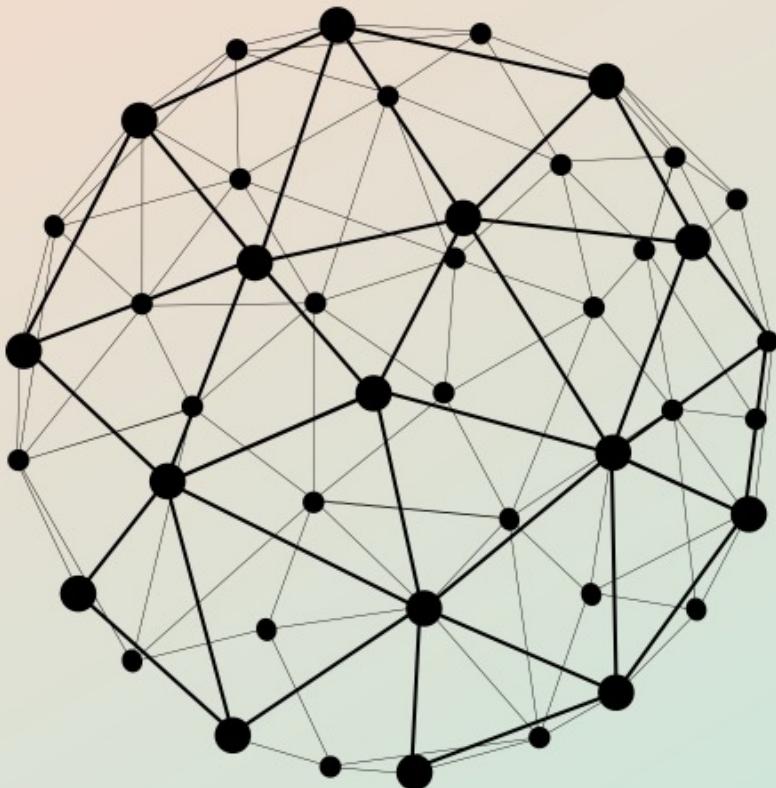
Spring injects the Vehicle bean into the Person bean using Dependency Injection, ensuring seamless management of dependencies. Additionally, Spring guarantees that only one instance of the Vehicle bean is created and that it is instantiated first, since the Person bean depends on it.



What is Autowiring in Spring?

Autowiring is a feature in Spring that automatically injects dependencies.

- ✓ It eliminates the need for manual object creation using new keyword.
- ✓ Spring finds the right dependency and injects it automatically.
- ✓ Makes applications loosely coupled and easier to maintain.



Why Use Autowiring?

- ✓ Reduces boilerplate code – No need for manual new object creation.
- ✓ Improves code readability – Dependencies are automatically injected.
- ✓ Supports loose coupling – Easy to modify and test.
- ✓ Spring manages dependency resolution efficiently.

Ways to Autowire in Spring

Spring provides three main ways to perform autowiring:

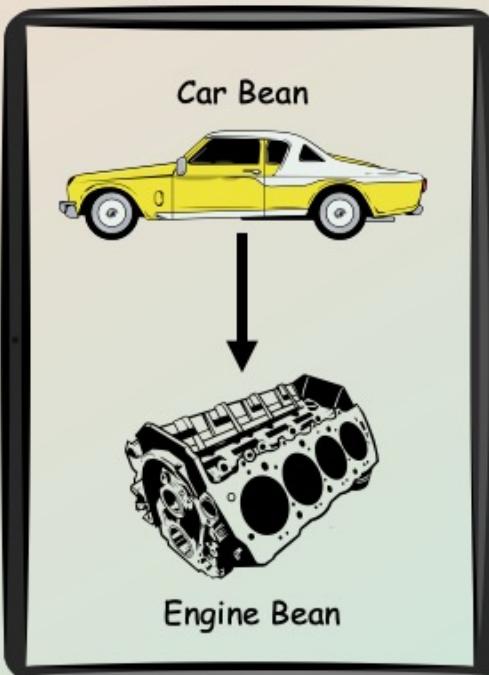
1. By Field Injection (@Autowired on field)
2. By Setter Injection (@Autowired on Setter Method)
3. By Constructor Injection (@Autowired on Constructor)

Autowiring using Field Injection (@Autowired on Field)

Spring looks for a matching bean type and injects it automatically. Simple and commonly used method.

```
@Component  
class Engine { }  
  
@Component  
class Car {  
  
    @Autowired  
    private Engine engine;  
}
```

Spring Context



While `@Autowired` by type (field injection) is convenient, it comes with drawbacks as mentioned below,

- ✓ When using field-based autowiring (`@Autowired`), dependencies cannot be easily mocked or replaced during unit testing.
- ✓ With field injection (`@Autowired` directly on fields), it's not immediately clear which dependencies a class requires. This makes the code harder to understand, maintain, and extend.
- ✓ Field injection does not allow dependencies to be declared as final, making the object mutable

`@Autowired(required = false)` will help to avoid the `NoSuchBeanDefinitionException` if the bean is not available during Autowiring process.

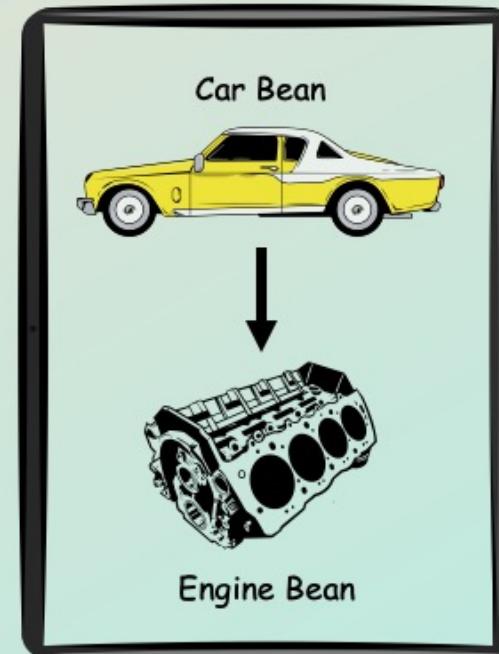
Autowiring using Setter Injection (@Autowired on Setter)

Spring injects the dependency using a setter method. Useful when the dependency is optional.

```
@Component  
class Engine { }  
  
@Component  
class Car {  
    private Engine engine;  
  
    @Autowired  
    public void setEngine(Engine engine) {  
        this.engine = engine;  
    }  
}
```

- ✓ Has same drawbacks as Field Injection
- ✓ Less commonly used compared to constructor injection.

Spring Context



Autowiring by Constructor Injection (@Autowired on Constructor)

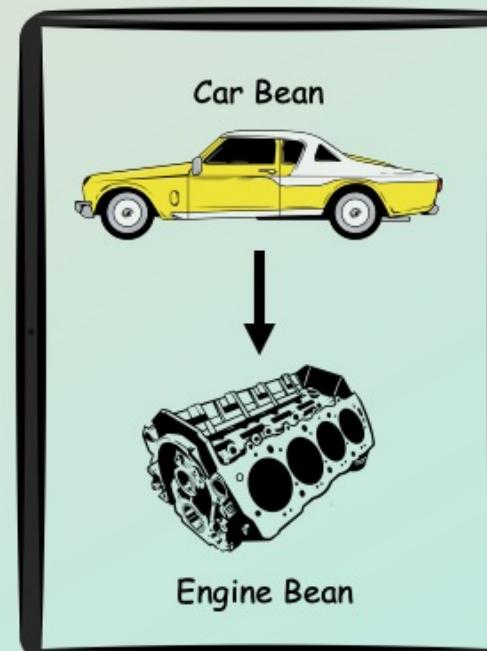
Spring injects dependencies through the constructor. Recommended for mandatory dependencies.



- ✓ Preferred approach in modern Spring applications.
- ✓ Works well with immutability and unit testing.

```
@Component  
class Engine { }  
  
@Component  
class Car {  
    private final Engine engine;  
  
    @Autowired  
    public Car(Engine engine) {  
        this.engine = engine;  
    }  
}
```

Spring Context



If multiple beans of the same type exist, we use @Qualifier to specify the exact bean. Helps when multiple beans of the same type exist. Avoids ambiguity in autowiring.

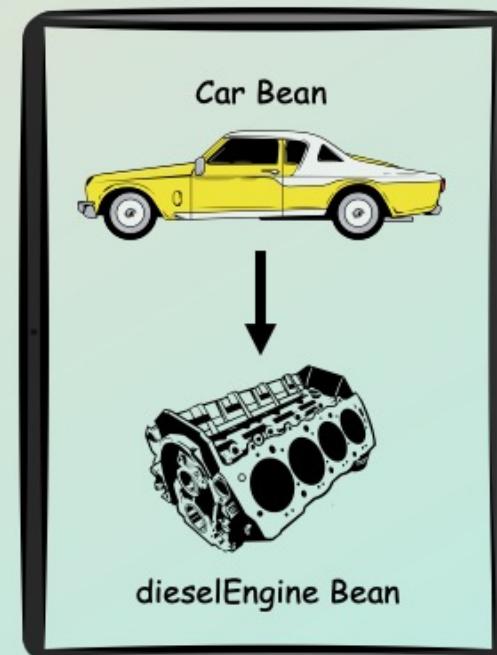
```
@Component("petrolEngine")
class PetrolEngine implements Engine { }

@Component("dieselEngine")
class DieselEngine implements Engine { }

@Component
class Car {
    @Autowired
    @Qualifier("dieselEngine")
    private Engine engine;
}
```

- ✓ @Qualifier is not limited to field injection; it works with constructor, setter, and method injection.
- ✓ Best practice: Use @Qualifier with constructor injection for required dependencies.

Spring Context



If @Primary is used, you do not need to use @Qualifier explicitly. Spring will automatically select the @Primary bean when multiple beans of the same type exist. However, there are cases where @Qualifier is still needed, even when @Primary is defined.

Annotation	When to Use?	How It Works
@Primary	When you have a default bean that should be used most of the time.	Spring automatically selects the @Primary bean if no specific qualifier is provided.
@Qualifier	When you need to explicitly specify which bean to use.	Overrides @Primary and allows selecting a specific bean when multiple exist.

Imagine you walk into a coffee shop that serves different types of coffee ☕

Example 1: @Primary (Default Choice)

The shop has multiple coffee types, but the barista always serves Espresso by default unless you ask for something specific. Here, Espresso is marked as @Primary, so whenever a customer asks for coffee without specifying a type, they get an Espresso.

```
@Component
@Primary
public class Espresso implements Coffee {}

@Component
public class Cappuccino implements Coffee {}

@Component
public class CoffeeShop {

    @Autowired
    private final Coffee coffee;
}
```



Example 2: @Qualifier (Explicit Selection)

Now, a customer wants a Cappuccino instead of the default Espresso. They specify their choice by saying: "I want a Cappuccino!"

Here, @Qualifier acts like the customer specifying their order, overriding the default choice.

```
@Component
@Primary
public class Espresso implements Coffee {}

@Component
public class Cappuccino implements Coffee {}

@Component
public class CoffeeShop {
    private final Coffee coffee;

    @Autowired
    public CoffeeShop(@Qualifier("cappuccino") Coffee coffee)
        this.coffee = coffee;
}
}
```



Understanding & Avoiding Circular dependencies

A circular dependency occurs when two or more beans depend on each other directly or indirectly. Spring cannot resolve dependencies and throws an `UnsatisfiedDependencyException`. This usually happens in constructor-based injection.

Example of Circular Dependency

Scenario:

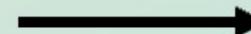
- Class Car depends on Class Engine.
- Class Engine depends on Class Car.
- Spring fails to resolve this dependency loop.

```
@Component
class Car {
    private final Engine eng;

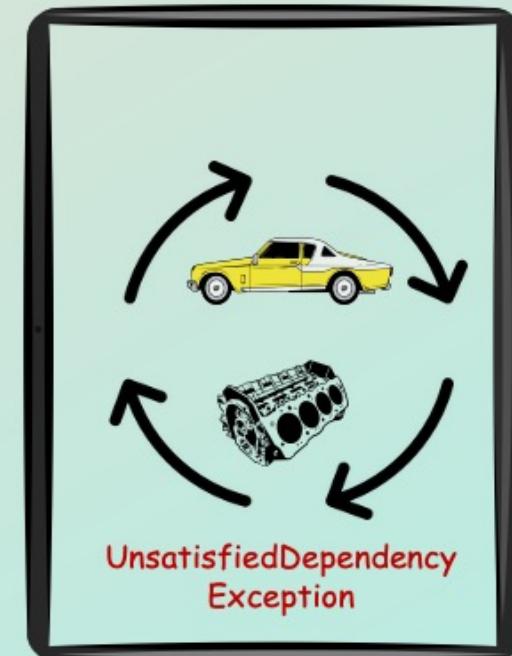
    @Autowired
    public Car(Engine eng) {
        this.eng = eng;
    }
}

@Component
class Engine {
    private final Car car;

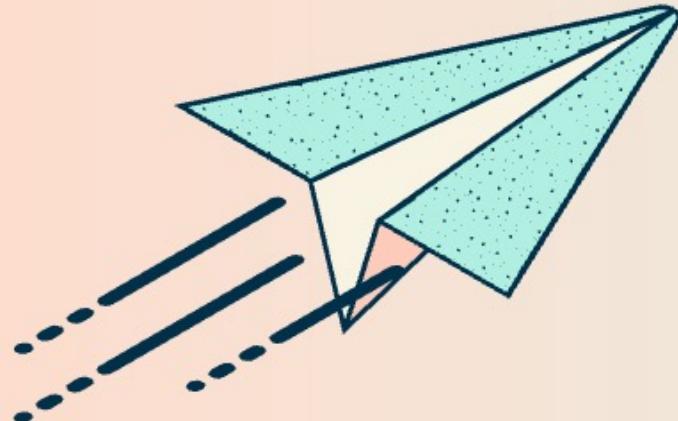
    @Autowired
    public Engine(Car car) {
        this.car = car;
    }
}
```



Circular dependency in
Spring Context



In Spring, a bean is an object that is managed by the Spring container. When you define a bean, you can also define its scope—which controls how and when Spring creates a new instance of that bean.



Spring provides five major bean scopes:

01

Singleton (Default)

02

Prototype

03

Request (for Web Apps)

04

Session (for Web Apps)

05

Application (for Web Apps)

Singleton is the default scope of a bean in Spring. This means that whenever a bean is referenced or autowired within the application, the same instance is returned each time.

Unlike the Singleton Design Pattern, which ensures only one instance per application, Spring's Singleton scope maintains one instance per bean definition within the Spring context. If multiple beans of the same type are declared, Spring will manage a separate singleton instance for each unique bean definition, rather than enforcing a single instance across the entire application.

```
@Component
@Scope("singleton") // Optional
public class MyService {
    public MyService() {
        System.out.println("MyService
instance created!");
    }
}
```

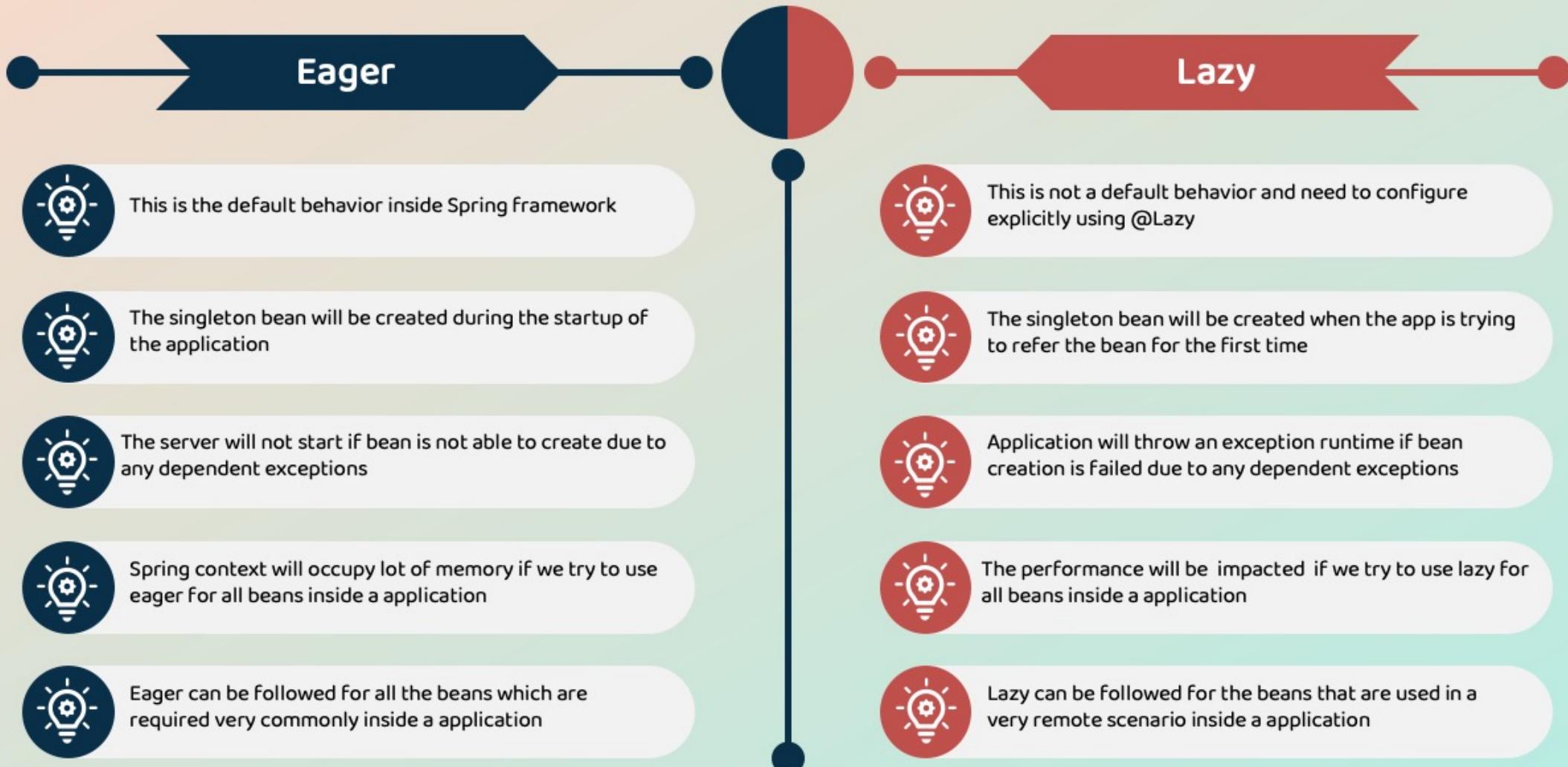
Use Singleton when you need a single shared instance across the application, such as a service managing a cache.

```
var context = new AnnotationConfigApplicationContext(AppConfig.class);
MyService service1 = context.getBean(MyService.class);
MyService service2 = context.getBean(MyService.class);

System.out.println(service1 == service2); // true (Both are the same instance)
```

Eager & Lazy instantiation

By default Spring will create all the singleton beans eagerly during the startup of the application itself. This is called Eager instantiation. We can change the default behavior to initialize the singleton beans lazily only when the application is trying to refer to the bean. This approach is called Lazy instantiation.



With prototype scope, every time we request a reference of a bean, Spring will create a new object instance and provide the same.

Prototype scope is rarely used inside the applications and we can use this scope only in the scenarios where you need a new instance each time, such as an object representing user input.

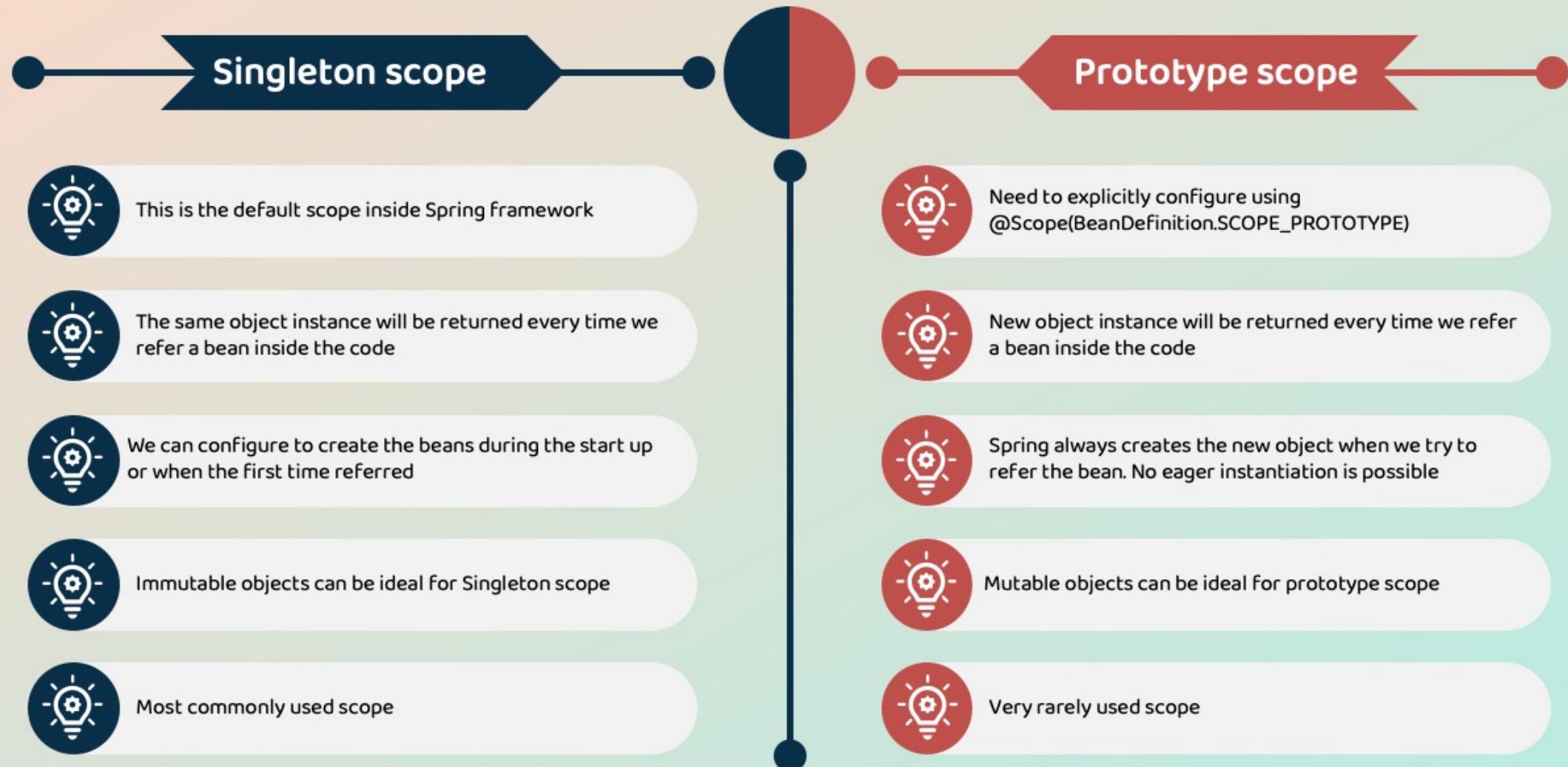
```
@Component
@Scope("prototype")
public class UserSession {
    private String sessionId;

    public UserSession() {
        this.sessionId = UUID.randomUUID().toString(); // Generate a unique ID
    }

    public String getSessionId() {
        return sessionId;
    }
}
```

```
UserSession user1 = context.getBean(UserSession.class);
UserSession user2 = context.getBean(UserSession.class);

System.out.println(user1 == user2); // false (Different instances)
```



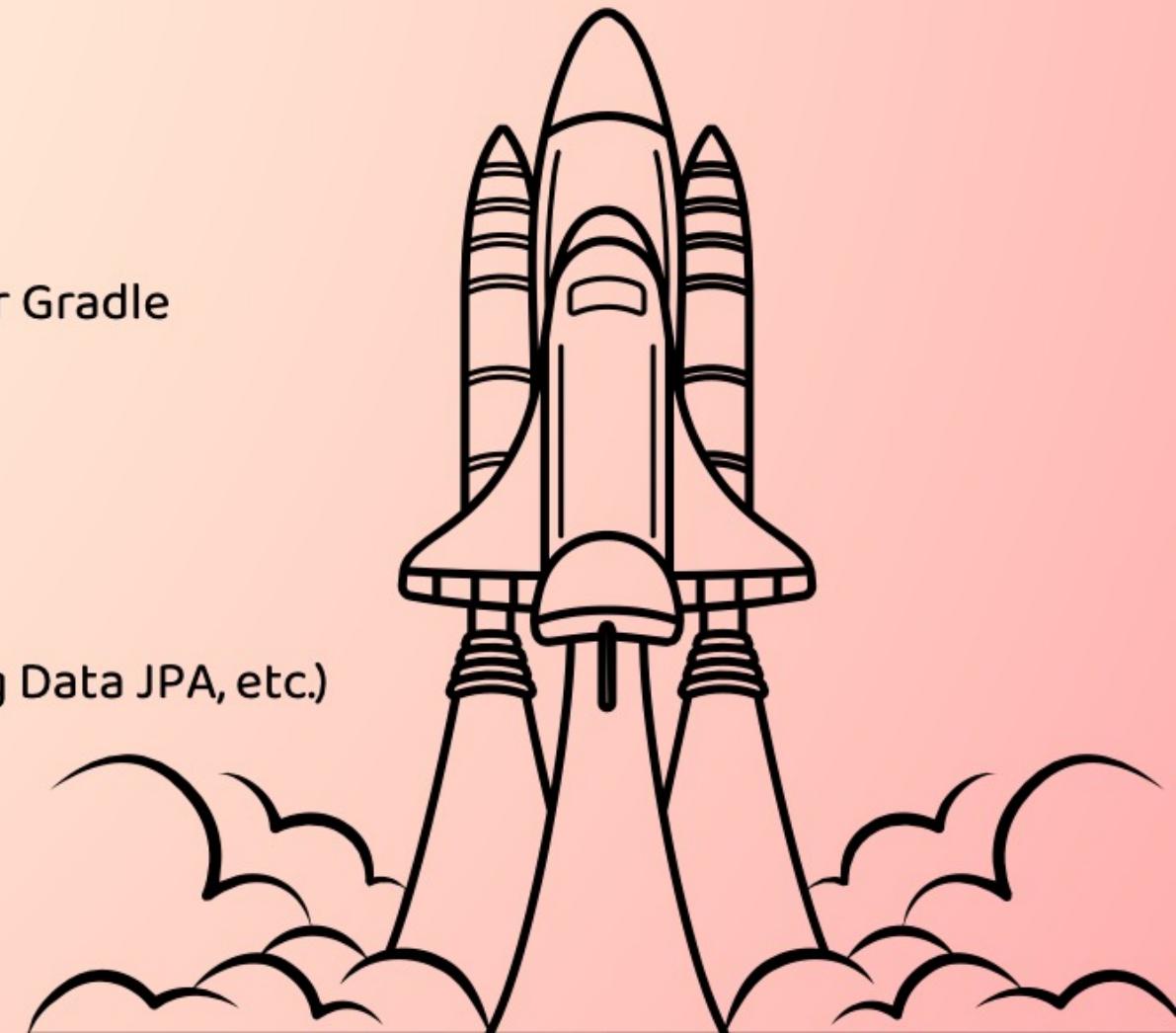


- 1 Spring Boot was introduced in April 2014 to reduce some of the burdens while developing a Java web application using Spring.
- 2 With SpringBoot, we can create Web Apps skeletons with in seconds or at least in 1-2 mins 😊. It helps eliminating all the configurations we need to do.
- 3 Spring Boot is now one of the most appreciated projects in the Spring ecosystem. It helps us to create Spring apps more efficiently and focus on the business code.
- 4 SpringBoot is a mandatory skill now due to the latest trends like Full Stack Development, Microservices, Serverless, Containers, Docker etc.

Spring Boot applications can be created using **Spring Initializr** which is a web-based tool that helps generate a Spring Boot project with required dependencies.

Steps:

- 1) Visit <https://start.spring.io/>
- 2) Choose build tool for the project either as Maven or Gradle
- 3) Choose Language: Java, Kotlin, or Groovy
- 4) Select the desired Spring Boot Version
- 5) Fill Project Metadata details
- 6) Add required Dependencies (e.g., Spring Web, Spring Data JPA, etc.)
- 7) Click Generate and download the project
- 8) Extract the ZIP file and open it in IntelliJ IDEA



What are Spring Boot Starters ?

Spring Boot Starters are ready-made dependency bundles that bring in all required libraries, configurations, and auto-settings for specific features.

Why Use Starters?

- They reduce manual setup of dependencies.
- They avoid version conflicts between libraries.
- They help you focus on coding instead of configuration.

How Do They Work?

Instead of adding multiple dependencies manually, you just add one Starter dependency in your pom.xml (Maven) or build.gradle (Gradle).



What are Spring Boot Starters ?

Example Without Starter (Manual Approach)

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-mvc</artifactId>
  </dependency>
  <dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-embed-core</artifactId>
  </dependency>
</dependencies>
```

Example With Starter

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Without starter projects, it will be too much work and can cause version conflicts!

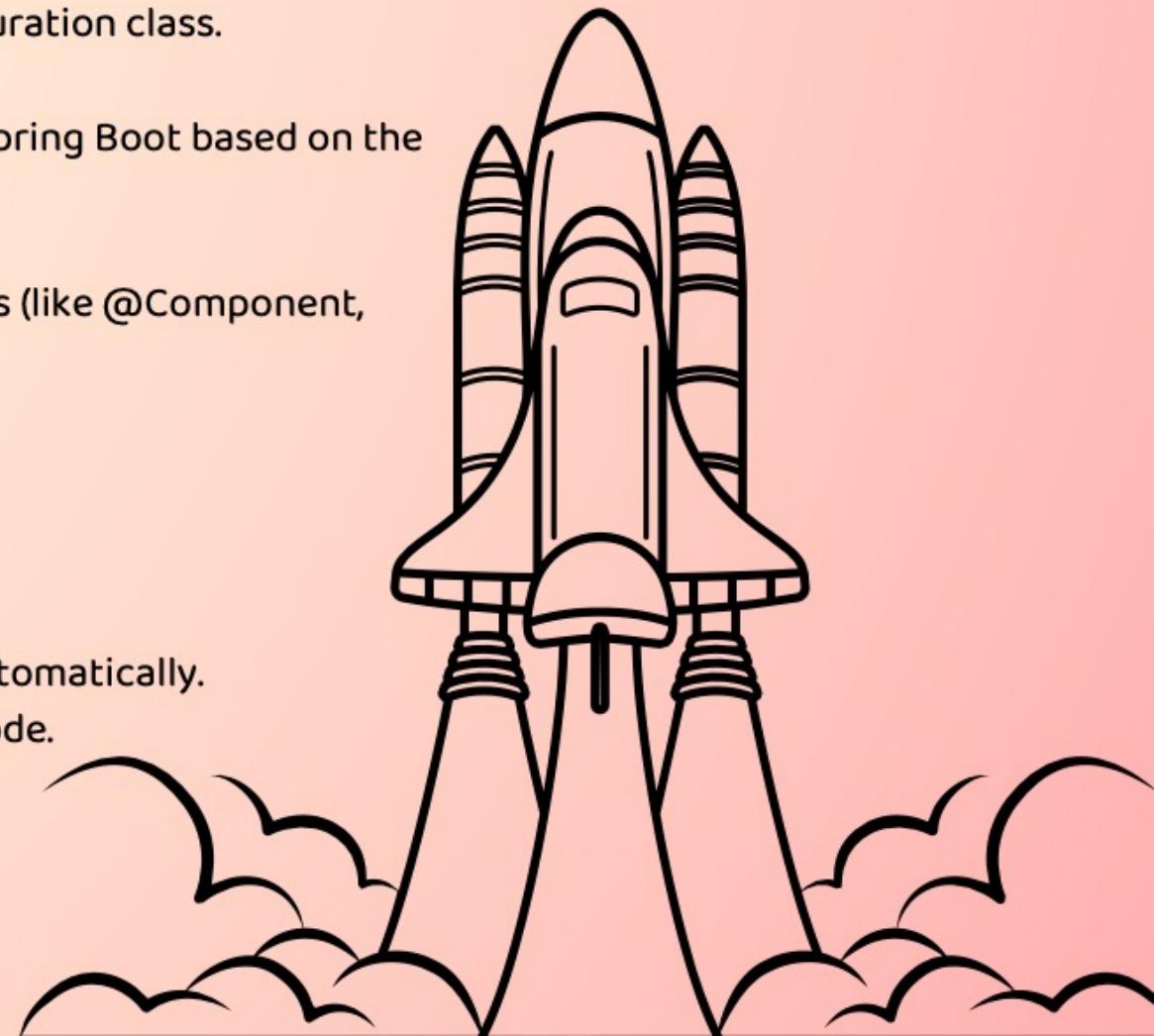
With starter projects, all the necessary libraries (Spring Web, MVC, Tomcat, etc.) will be automatically pulled

@SpringBootApplication is a shortcut annotation that combines three annotations:

1. `@SpringBootConfiguration` → Marks this class as a configuration class.
2. `@EnableAutoConfiguration` → Automatically configures Spring Boot based on the dependencies.
3. `@ComponentScan` → Scans and registers Spring components (like `@Component`, `@Service`, etc.).

Why Use @SpringBootApplication?

- Saves time – No need for long XML configuration.
- Auto-configures – Detects dependencies and configures them automatically.
- Simplifies Spring Boot setup – Everything works with minimal code.



Auto-Configuration in Spring Boot automatically configures your application based on the dependencies you add. In simple words, you don't have to manually set up configurations. Spring Boot does it for you!

How Does Auto-Configuration Work?

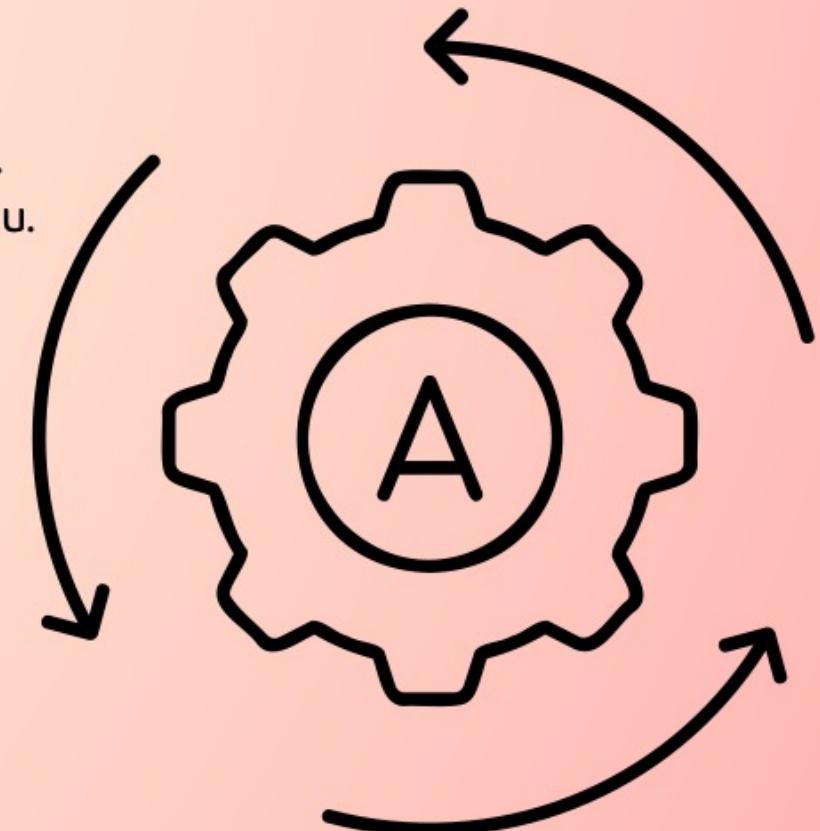
1. You add a Spring Boot Starter dependency (e.g., `spring-boot-starter-web`).
2. Spring Boot automatically detects what you need and configures it for you.
3. You can override the default settings if needed.

For example, If `spring-boot-starter-web` is present, Spring Boot assumes:

- The app is a web application.
- It starts an embedded Tomcat server on port 8080.
- It creates a default DispatcherServlet for handling requests.

Few of the default values assumed during Auto Configuration:

```
server.port=8080
server.servlet.context-path=/
logging.level.root=INFO
```



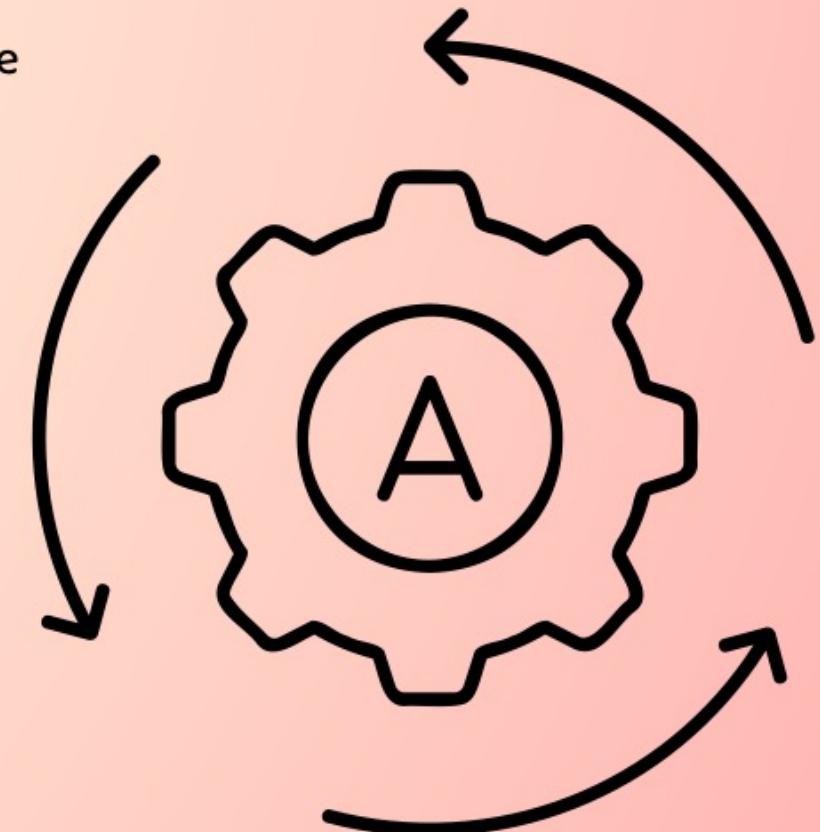
If you were using **traditional Spring**, you would have to manually configure **a web server, database connections**, and other settings.

Auto-Configuration is enabled by default in Spring Boot. It is controlled by the `@SpringBootApplication` annotation. This annotation includes `@EnableAutoConfiguration`, which tells Spring Boot to configure everything automatically.

Mentioning `debug=true` will print the Autoconfiguration report on the console.

If you want to disable auto-configuration for specific components, you can use:

```
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
```



Spring and Spring Boot allow developers to build powerful web applications. These applications can be classified into two major types. Spring MVC is the key differentiator between these two approaches.

Spring MVC-Based Web Apps

- Web Apps which holds UI elements like HTML, CSS, JS and backend logic
- Here the App is responsible to fully prepare the view along with data in response to a client request
- Spring Core, **Spring MVC**, SpringBoot, Spring Data, Spring Rest, Spring Security will be used

REST API-Based Web Applications

- Web Apps which holds only backend logic. These Apps send data like JSON to separate UI Apps built based on Angular, React etc.
- Here the App is responsible to only process the request and respond with only data ignoring view.
- Spring Core, SpringBoot, Spring Data, Spring Rest, Spring Security will be used

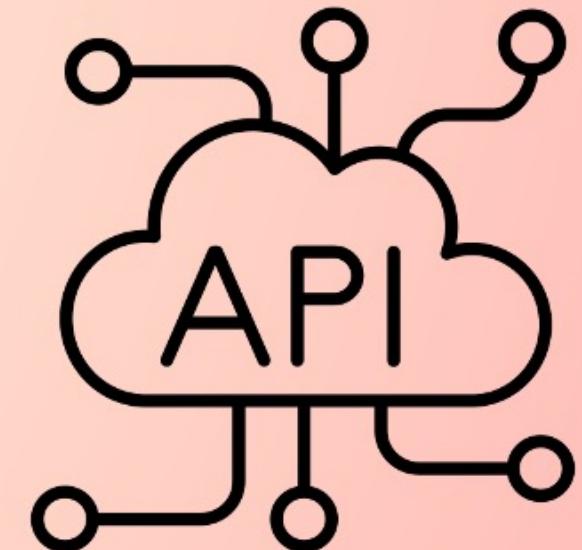
What is a REST API?

REST (Representational State Transfer) is a way for applications to communicate over HTTP. A REST API:

- Allows clients (e.g., web, mobile apps) to interact with your application.
- Returns JSON or XML responses.

REST APIs use HTTP methods for different actions:

HTTP Method	Action
GET	Fetch data
POST	Create new data
PUT	Full update (replaces the entire resource)
PATCH	Partial update (modifies only specified fields)
DELETE	Remove data



In Spring Boot, a REST API is created using `@RestController`. The `@RestController` annotation in Spring Boot is a specialized version of `@Controller` that simplifies the development of RESTful web services.

`@RestController = @Controller + @ResponseBody`

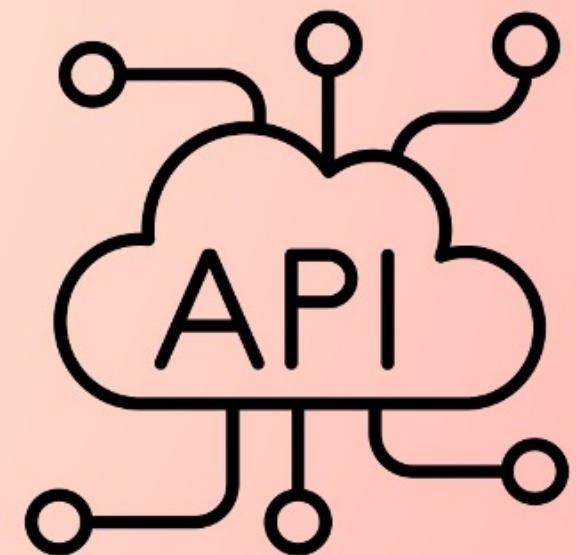
This means that:

1. The class is treated as a controller that handles HTTP requests.
2. Each method automatically serializes the return value into JSON (or XML) and sends it in the HTTP response.

Example:

```
@RestController
@RequestMapping("/api/v1/users")
public class UserController {

    @GetMapping
    public List<String> getUsers() {
        return List.of("Alice", "Bob", "Charlie");
    }
}
```



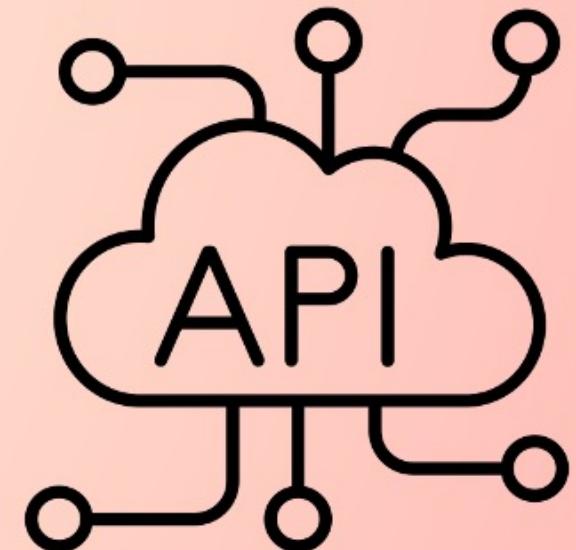
Use nouns, not verbs while defining REST API paths,

Good Examples:

GET api/v1/users	→ Get all users
GET api/v1/users/{id}	→ Get a specific user
POST api/v1/users	→ Create a new user
PUT api/v1/users/{id}	→ Update user details
DELETE api/v1/users/{id}	→ Delete a user

Avoid bad practices

GET api/v1/getUsers	✗
POST api/v1/createUser	✗
DELETE api/v1/removeUser	✗



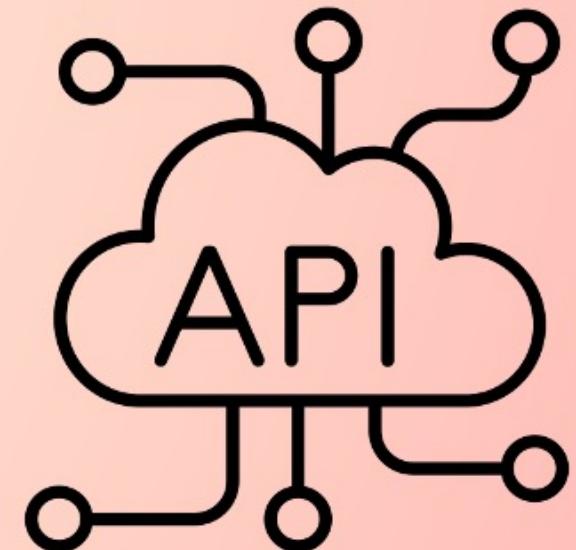
When you build REST APIs, over time, you might need to make changes like:

- Adding new fields.
- Changing request/response formats.
- Modifying business logic.
- Deprecating old features.

To avoid breaking existing clients, we use **API Versioning**

There are multiple ways to version REST APIs. The most popular ones are:

1. URI Versioning (Path Versioning)
2. Request Parameter Versioning
3. Header Versioning
4. Media Type Versioning (Content Negotiation)



URI Versioning (Path Versioning)

```
GET /api/v1/users  
GET /api/v2/users
```

Request Parameter Versioning

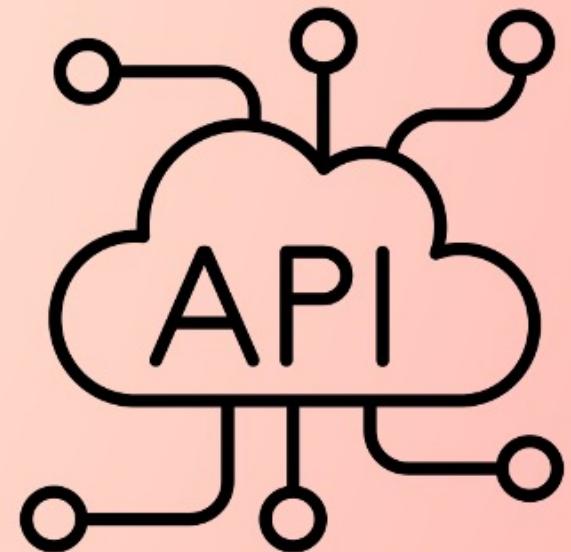
```
GET /api/users?version=1  
GET /api/users?version=2
```

Header Versioning

```
GET /api/users  
Headers:  
X-API-VERSION: 1
```

Media Type Versioning (Content Negotiation)

```
GET /api/users  
Accept: application/vnd.company.v1+json
```



Return proper HTTP status codes. Avoid returning 200 OK for everything

HTTP Method	Purpose	Status code
GET	Fetch data	200 OK
POST	Create new data	201 Created or 202 Accepted
PUT	Full update	200 OK or 204 No Content
PATCH	Partial update	200 OK or 204 No Content
DELETE	Remove data	200 OK or 204 No Content

Additional exception status codes

- 400 Bad Request → If the request is invalid
- 401 Unauthorized → When authentication is required but missing or incorrect.
- 403 Forbidden → When access is denied despite authentication.
- 404 Not Found → If the resource does not exist.
- 405 Method Not Allowed → If the HTTP method is not allowed on the resource.
- 500 Internal Server Error → For unexpected server errors.
- 503 Service Unavailable → If the server is temporarily overloaded or down.

What is H2 Database?

H2 is a lightweight, fast, and in-memory database used for:

- Development & Testing (because it resets on restart)
- Embedded Mode (no installation needed)
- Standalone Mode (can store data on disk)

Why Use H2 in Spring Boot?

- No need to install – It runs in memory.
- Fast & lightweight – Perfect for testing.
- Supports SQL – Similar to MySQL, PostgreSQL.
- Provides a Web Console – Easy to view data.

Adding H2 to Your Spring Boot Project – By adding below dependencies in pom.xml, Spring Boot automatically detects

H2 DB and configures it

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```



Configuring H2 Database in application.properties

```
# H2 Database Configuration
spring.datasource.url=jdbc:h2:mem:testdb # In-memory database
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# Enable H2 Console (for viewing tables & data)
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```



What does above properties do?

- `jdbc:h2:mem:testdb` → Uses an in-memory database (testdb).
- `sa` (default username), empty password.
- `H2 Console enabled` → View database in the browser.
- `H2 Dialect` → Configures Hibernate for H2.

Running H2 Database in Spring Boot

1. Run your Spring Boot application.
2. Open a browser and go to: <http://localhost:8080/h2-console>
3. Enter database details:
 - JDBC URL: jdbc:h2:mem:testdb
 - Username: sa
 - Password: (leave empty)
4. Click Connect

 Now, you can execute SQL queries and view tables!

H2 Persistent Mode

By default, H2 resets data on restart (because it's in-memory). To keep data between restarts, change the URL:

```
spring.datasource.url=jdbc:h2:file:~/eazystore;AUTO_SERVER=true
```



Spring Boot auto-creates DB schema & inserts data from SQL scripts.

Default script locations:

- Schema (DDL) → optional:classpath*:schema.sql
- Data (DML) → optional:classpath*:data.sql

Custom locations:

- Set using `spring.sql.init.schema-locations` and `spring.sql.init.data-locations`

The optional: prefix means that the application will start even when the files do not exist. To have the application fail to start when the files are absent, remove the optional: prefix.

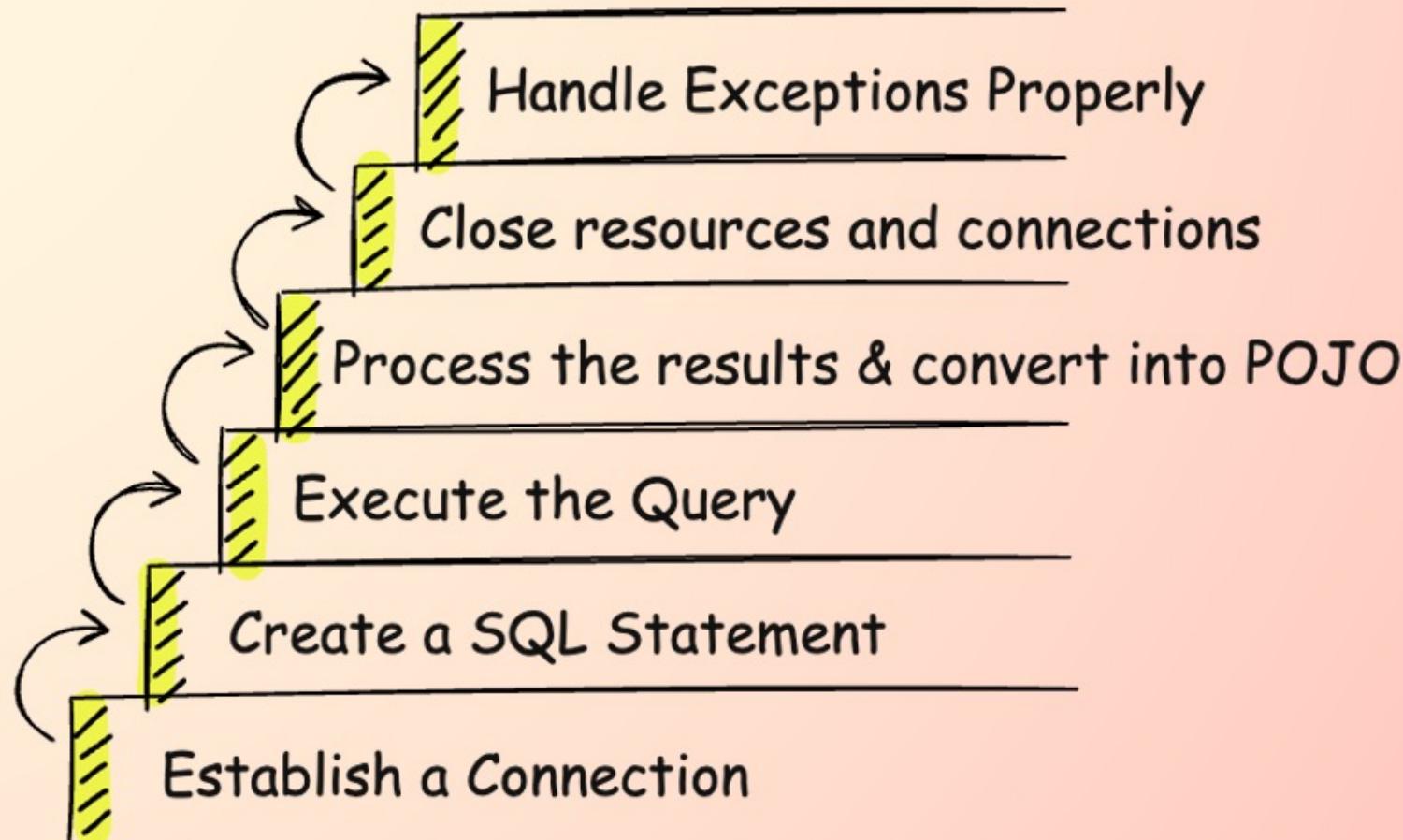
When does initialization run?

- By default: embedded (Only for in-memory databases)
- Force always: Set `spring.sql.init.mode=always`
- Disable: Set `spring.sql.init.mode=never`



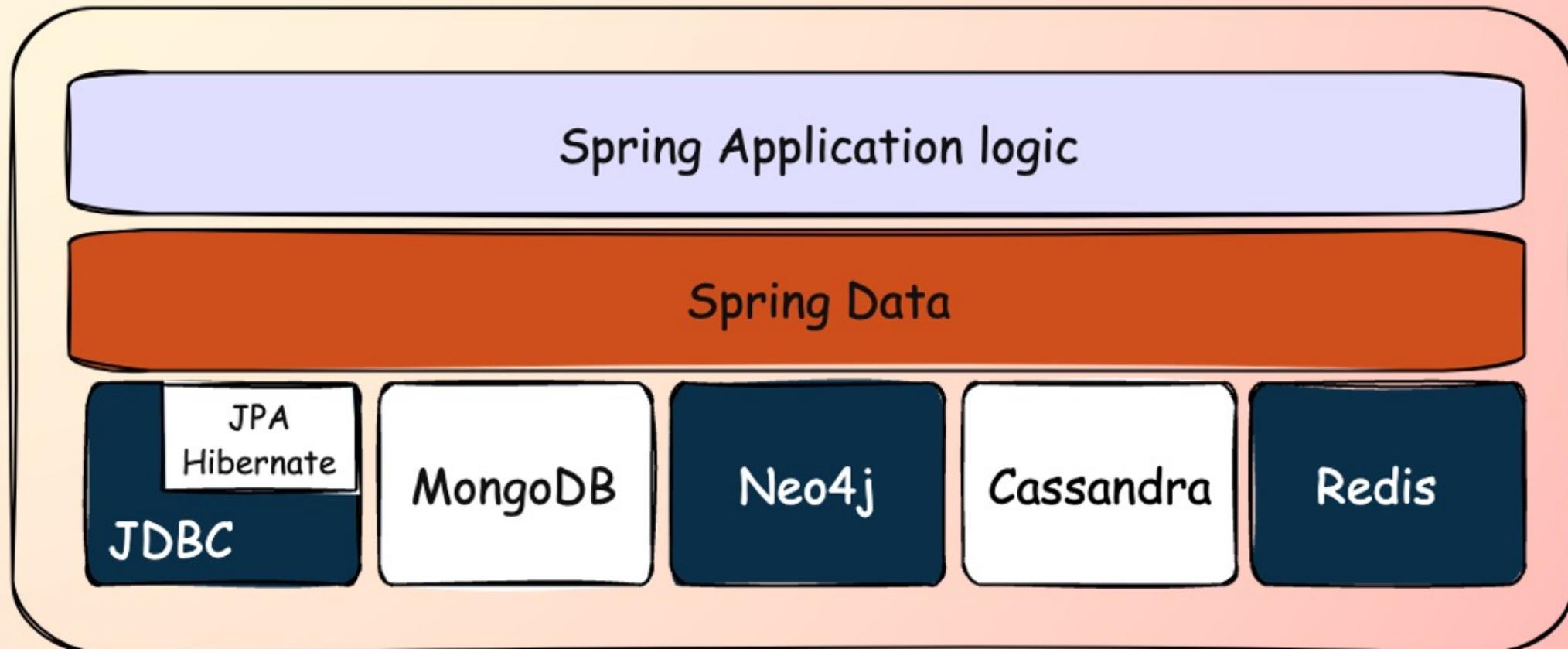
Typical Steps while working with a DB using Java

All the below steps needs to be managed by Developer manually when not using any frameworks like Spring Data JPA,



Spring Data is a part of the Spring Framework that simplifies database access in Java applications. It provides an easy and consistent way to interact with databases without writing boilerplate code.

Spring Data is an umbrella project that includes several modules to work with different types of databases (relational, NoSQL, etc.).



Why Use Spring Data?

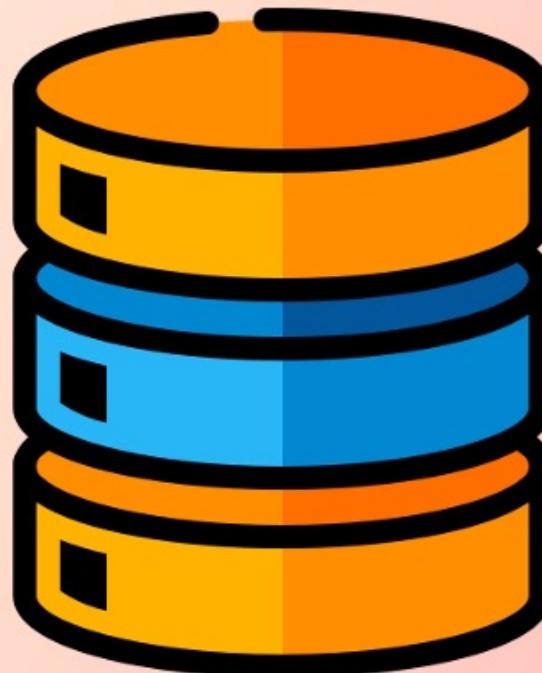
Spring Data helps you to:

- Reduce boilerplate code for database operations
- Simplify CRUD (Create, Read, Update, Delete) operations
- Work seamlessly with Spring Boot

Spring Data with Different Databases

Spring Data supports:

- Relational Databases (MySQL, PostgreSQL, Oracle)
- NoSQL Databases (MongoDB, Redis, Cassandra)
- Graph Databases (Neo4j)



Spring Data and Spring Data JPA are closely related, but they are not the same. Let's break it down in simple terms so you can understand the differences clearly.



- ✓ **Spring Data** is an umbrella project that simplifies working with databases.
- ✓ It provides common functionalities for different types of databases, including SQL, NoSQL, Redis, Neo4j etc.
- ✓ Think of Spring Data as a toolbox that contains different tools for working with databases
- ✓ **Spring Data JPA** is a subproject of Spring Data that focuses on JPA (Java Persistence API)
- ✓ It helps you interact with relational databases like MySQL, PostgreSQL, Oracle, and H2 without writing complex SQL queries.
- ✓ Spring Data JPA is a part of Spring Data but specifically for JPA-based databases. It uses Hibernate as the default JPA provider to manage the database.

The most commonly used sub module of Spring Data is **Spring Data JPA**, which simplifies working with **relational databases**

No matter which persistence technology your application uses, Spring Data offers a unified set of interfaces that you can extend to define its persistence capabilities.

At the core of Spring Data's repository abstraction is the **Repository** interface, which serves as the foundation for building data access layers efficiently.

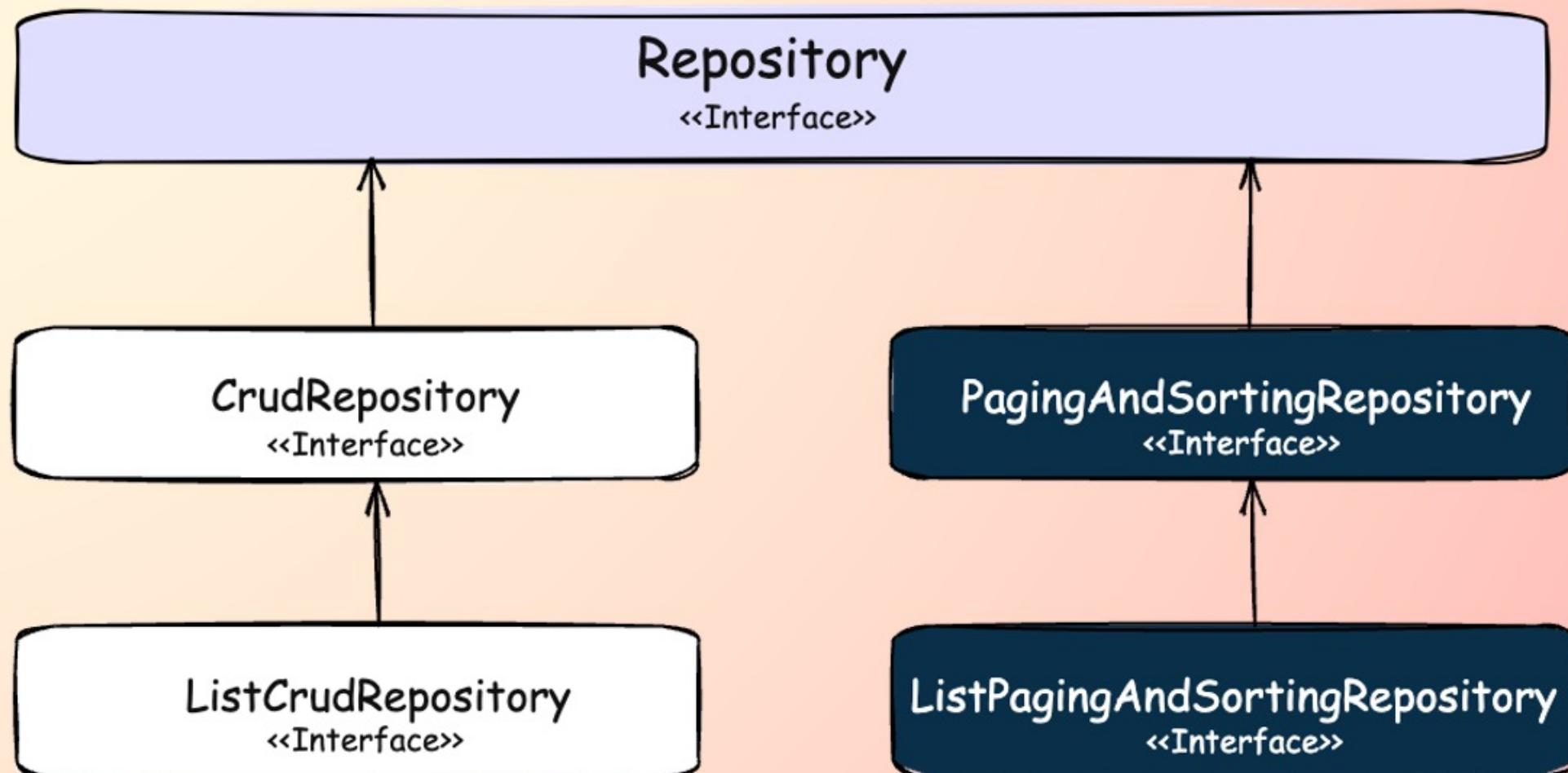
The **Repository** interface is the most abstract contract in Spring Data. Extending it marks your interface as a Spring Data repository, but it provides no predefined operations like saving or retrieving records. It acts purely as a marker interface without declaring any methods.

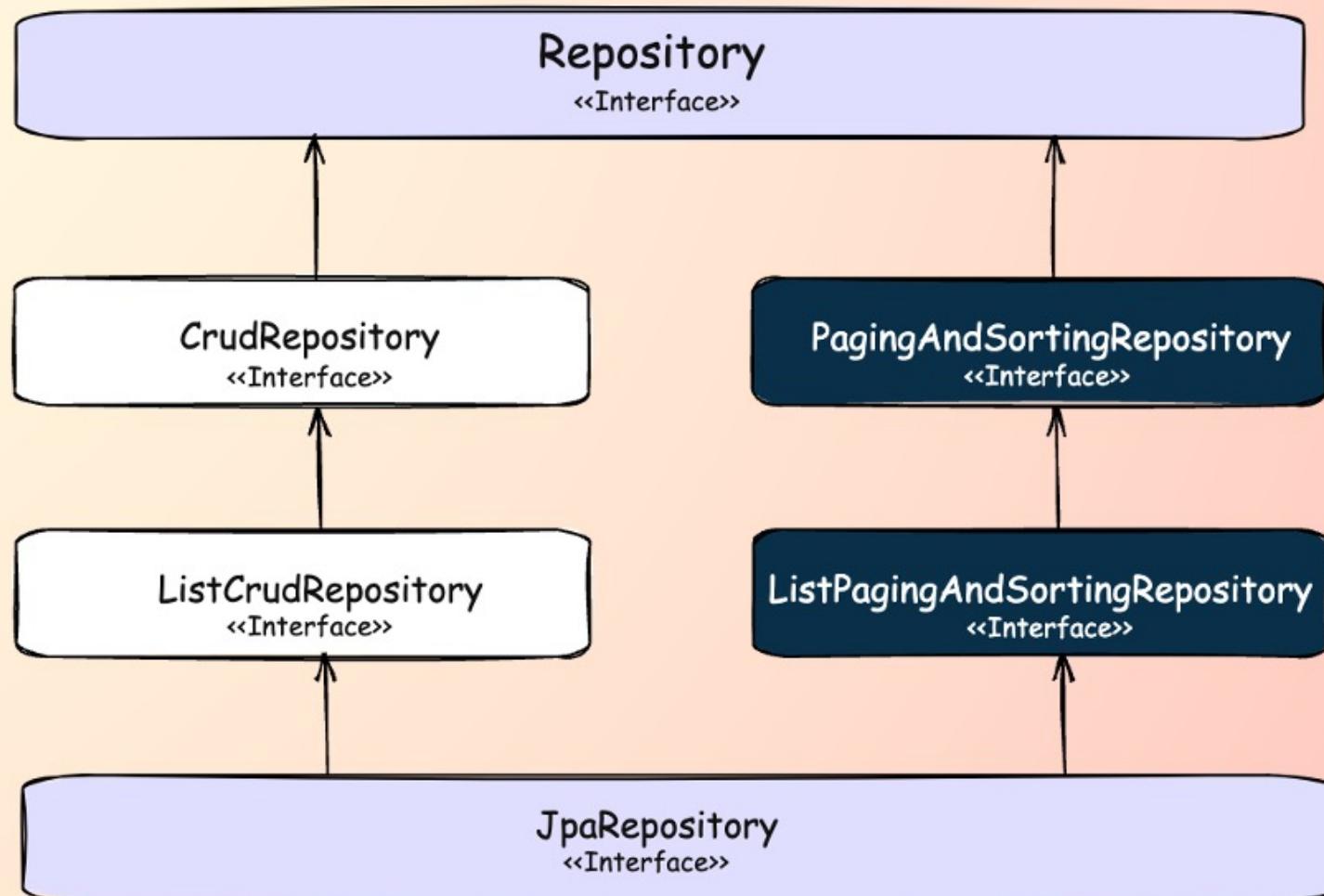
CrudRepository is the simplest Spring Data contract that provides basic persistence operations—create, retrieve, update, and delete. Extending it enables these functionalities in your application. **ListCrudRepository** extends **CrudRepository**, returning **List** instead of **Iterable** where applicable.

PagingAndSortingRepository provides methods for retrieving entities with pagination and sorting. **ListPagingAndSortingRepository** extends it, returning **List** instead of **Iterable** where applicable.

Spring Data important interfaces

To implement repositories in Spring Data, extend one of its core interfaces: CrudRepository, ListCrudRepository, PagingAndSortingRepository, or ListPagingAndSortingRepository. Each provides different levels of persistence functionality.





Choose JpaRepository from Spring Data JPA, when working with relational databases like MySQL, PostgreSQL, or Oracle, where data is structured in tables with relationships.

Quick Tips

It's important to distinguish between the `@Repository` annotation and the Spring Data Repository interface.

Spring Data follows the Interface Segregation Principle, providing multiple repository interfaces that extend one another. This allows applications to adopt only the functionality they need, avoiding unnecessary complexity.

Different Spring Data modules offer technology-specific repository contracts. For instance:

- Spring Data JPA provides `JpaRepository` for relational databases.
- Spring Data MongoDB offers `MongoRepository` for working with MongoDB.

By extending the appropriate interface, applications can leverage Spring Data's powerful abstraction while maintaining flexibility and efficiency.



Step 1: Add Dependencies

Include the dependencies of Spring Data JPA and relational DB like H2, MySQL etc.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Step 2: Configure Database details in application.properties

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.show-sql=true
```



1. Entity

What it does:

- Represents a table in the database.
- Each instance of the entity is a row in the table.

```
import jakarta.persistence.*;  
  
@Entity  
@Table(name = "users")  
public class User {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    @Column(name = "name" )  
    private String name;  
  
    @Column(name = "email" )  
    private String email;  
  
    // Getters and Setters  
}
```

- **@Entity** → Marks a class to represents a table in the database.
- **@Table(name = "users")** → Defines the table name (Optional when table name and class name is same)
- **@Id** → Marks the primary key.
- **@GeneratedValue** → Allow to configure the primary key value strategy
- **@Column(name= "name")** → Defines the column name (Optional when column name and field name is same)

2. Repository

What it does:

- Allows you to interact with the database without writing queries.
- Spring Data JPA provides support to perform CRUD operations without writing any code

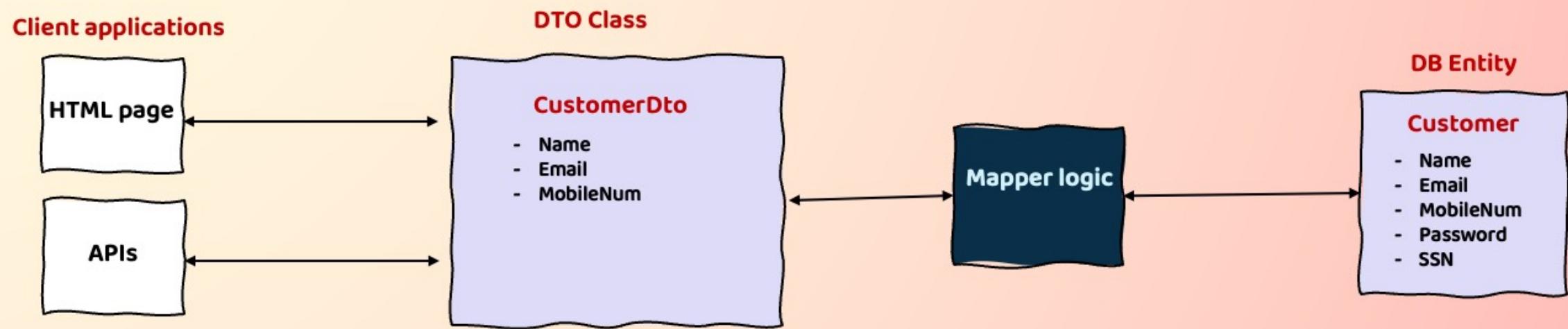
```
import org.springframework.data.jpa.repository.JpaRepository;  
  
@Repository // Optional  
public interface UserRepository extends JpaRepository<User, Long> {  
  
}
```

- `JpaRepository<User, Long>` → Provides ready-made CRUD supported methods like `save()`, `findById()`, `delete()`, etc.
- `UserRepository` can be injected into the service layer to trigger the DB related operations

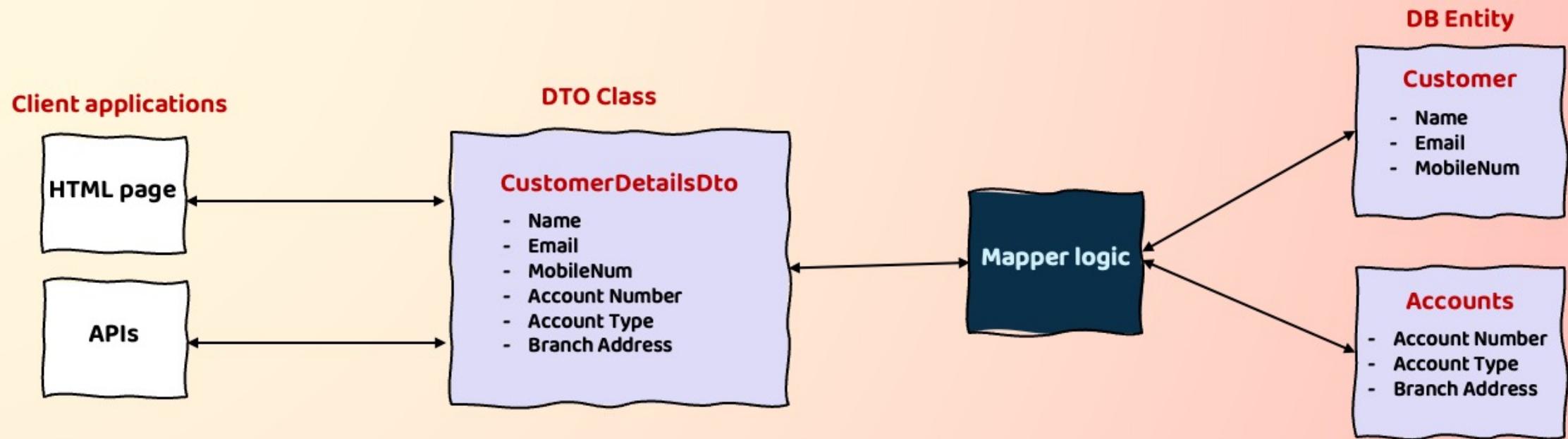
- If your repositories and entities are in the different package as `@SpringBootApplication`, then you should mention `@EnableJpaRepositories`, `@EntityScan` annotations along with the package details.

DTO (Data Transfer Object) pattern

The Data Transfer Object (DTO) pattern is a design pattern that allows you to transfer data between different parts of your application. DTOs are simple objects that contain only data, and they do not contain any business logic. This makes them ideal for transferring data between different layers of your application, such as the presentation layer and the data access layer.



DTO (Data Transfer Object) pattern



Here are some of the benefits of using the DTO pattern:

Reduces network traffic: DTOs can be used to batch up multiple pieces of data into a single object, which can reduce the number of network requests that need to be made. This can improve performance and reduce the load on your servers.

Encapsulates serialization: DTOs can be used to encapsulate the serialization logic for transferring data over the wire. This makes it easier to change the serialization format in the future, without having to make changes to the rest of your application.

Decouples layers: DTOs can be used to decouple the presentation layer from the data access layer. This makes it easier to change the presentation layer without having to change the data access layer.

What is CORS?

CORS is a security feature built into web browsers. It prevents a website from making requests to a different domain unless explicitly allowed.

Example of a CORS Issue

Imagine you have:

- A frontend running at `http://localhost:5173` (React)
- A backend API running at `http://localhost:8080` (Spring Boot)

If your frontend tries to fetch data from the backend below CORS error will be thrown by the browser:

CORS Error:

```
Access to fetch at 'http://localhost:8080/api/users' from origin
'http://localhost:3000' has been blocked by CORS policy.
```

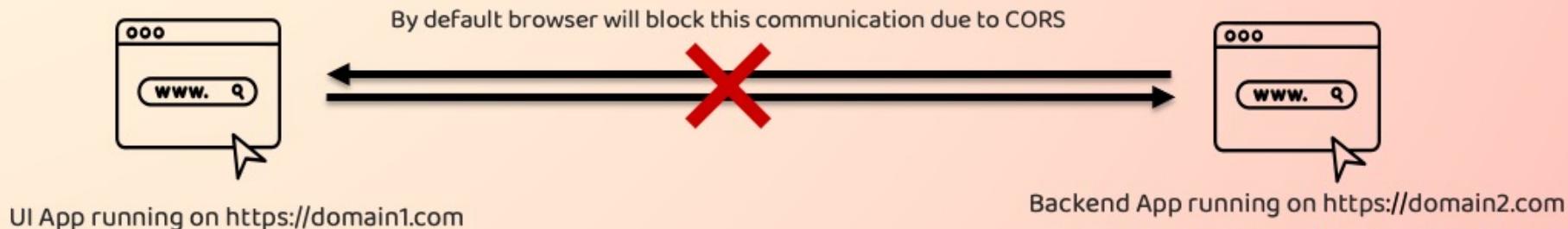


CORS is not a security issue/attack but the default protection provided by browsers to stop sharing the data/communication between different origins.

When does two origins consider different ?

Two origins consider different if they have:

- a different scheme (HTTP or HTTPS)
- a different domain
- a different port



How to Fix CORS?

If a web application's UI is deployed on one server and needs to communicate with a REST service hosted on another, we can enable this interaction by making changes in the backend app. Backend can decide from which origins, it want to communicate.

How to Fix CORS using Spring Boot?

Option 1: Using `@CrossOrigin` (Simple & Quick):

Enable CORS on a Specific Endpoint

```
@RestController
@RequestMapping("/api/v1")
public class UserController {

    @CrossOrigin(origins = "http://localhost:5173")
    @GetMapping("/users")
    public String getUsers() {
        return "List of users";
    }
}
```

Enable CORS for the Entire Controller

```
@RestController
@RequestMapping("/api/v1")
@CrossOrigin(origins = "http://localhost:5173")
public class UserController {

    @GetMapping("/users")
    public String getUsers() {
        return "List of users";
    }
}
```

Option 2: Using a Filter

If you have many controllers, it's better to configure CORS globally.

```
@Configuration
public class CorsConfig {

    @Bean
    public CorsFilter corsFilter() {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowedOrigins(Arrays.asList("http://localhost:5173"));
        config.setAllowedMethods(Collections.singletonList("*"));
        config.setAllowedHeaders(Arrays.asList("Content-Type"));
        config.setAllowCredentials(true);

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", config);
        return new CorsFilter(source);
    }
}
```

Option 3: Using Spring Security (will be discussed in Spring Security sections)

When CORS is enabled, Spring Boot adds headers like below:

Header	Description
Access-Control-Allow-Origin	Specifies which origins are allowed (e.g., <code>http://localhost:5173</code>)
Access-Control-Allow-Methods	Lists allowed HTTP methods (GET, POST, PUT, DELETE)
Access-Control-Allow-Headers	Specifies allowed request headers
Access-Control-Allow-Credentials	Allows sending cookies (e.g., JWT tokens)

When NOT to Use CORS config?

If your frontend and backend run on the SAME domain, CORS is not needed.

Example:

- `https://mywebsite.com` (frontend)
- `https://mywebsite.com/api` (backend) → No CORS required.

React events work similarly to standard HTML events (like onClick, onChange, etc.), but with some key differences. React uses synthetic events, which wrap native browser events to make them work consistently across all browsers.

How to Handle Events in React

In React, we handle events using event handler functions. These functions are passed as props to React elements.

```
function App() {
  // Event handler function
  function handleClick() {
    alert("Button clicked!");
  }

  return (
    <div>
      <button onClick={handleClick}>Click Me</button>
    </div>
  );
}
```

The function handleClick gets executed when the button is clicked.

Instead of writing onClick="handleClick()" like in plain HTML, React uses {handleClick} without parentheses.



Why writing onClick={handleClick()} is not allowed ?

In HTML, we write `onclick="handleClick()"` because it is an inline event handler written as a string in the HTML attribute. But in React, event handlers are written as JavaScript expressions inside curly braces {}

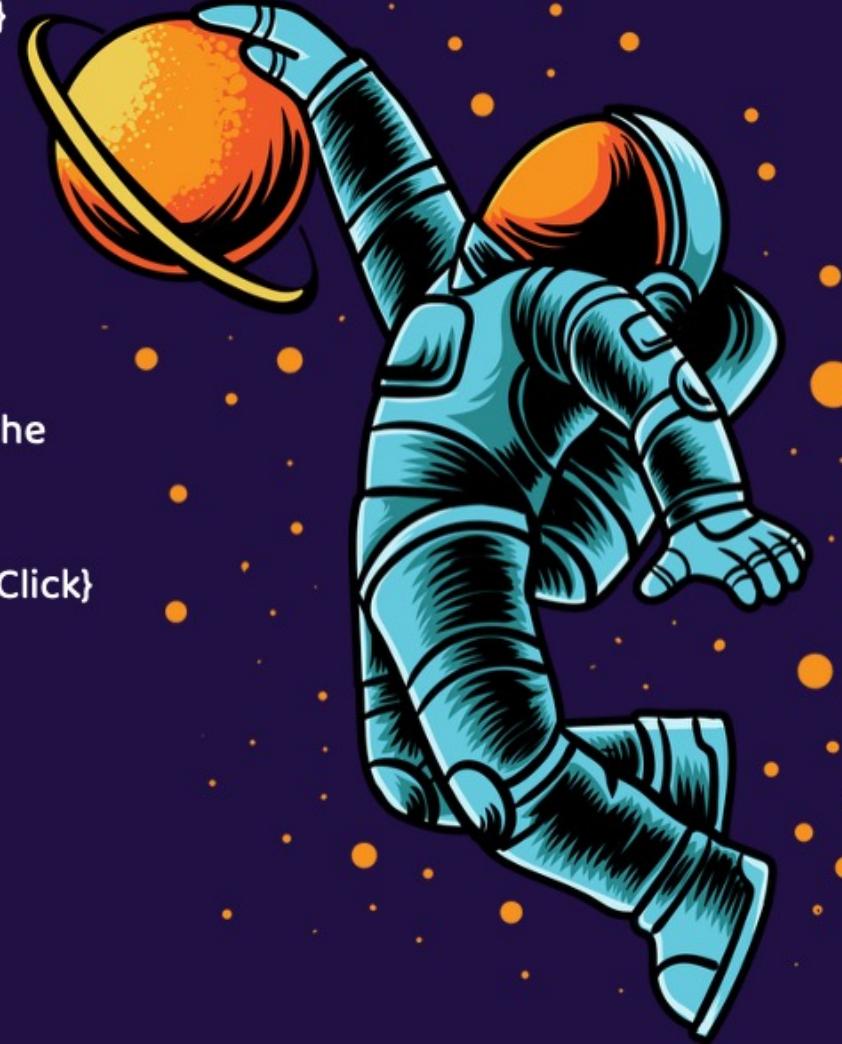
Writing `handleClick()` inside JS will immediately call `handleClick` when the component renders, instead of waiting for the user to click the button.

What Happens if You Write onClick="handleClick()" in React ?

React will treat "`handleClick()`" as a string, not as an actual function. It will not execute the function as expected.

Instead, React expects a reference to a function, which is why we use: `onClick={handleClick}`

Here, `handleClick` is passed without parentheses, meaning React knows it's a function and will call it when the button is clicked.



Passing Arguments in Event Handlers

If you want to pass additional arguments to your event handler, you can use an arrow function.

```
function App() {  
  function handleClick(name) {  
    alert(`Hello, ${name}!`);  
  }  
  
  return (  
    <button onClick={() => handleClick("Madan")}>  
      Greet Me  
    </button>  
  );  
}
```

Here, we use an arrow function inside onClick so we can pass "Madan" as an argument.



Event Object in React

React provides an event object containing details about the event (like mouse position, key pressed, etc.)

```
function App() {  
  function handleClick(event) {  
    console.log(event); // Logs the event object  
  }  
  
  return (  
    <button onClick={handleClick}>Click Me</button>  
  );  
}
```

Open your browser console and click the button—you'll see details about the click event!

Though you're not calling `handleClick(event)` yourself, React automatically calls your `handleClick` function with an event object when the event occurs



Does React Still Pass the Event Object If you Pass Custom Arguments?

No, if you pass custom arguments alone in an event handler, React does not automatically pass the event object.

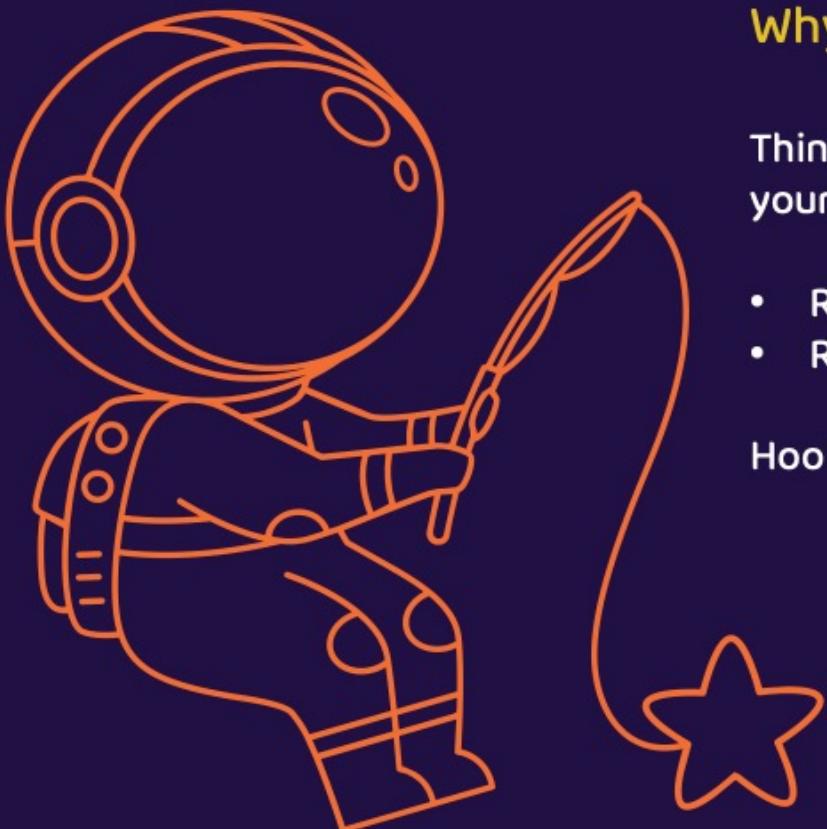
```
function App() {  
  function handleClick(name, event) {  
    console.log(`Hello, ${name}!`);  
    console.log(event);  
  }  
  
  return (  
    <button onClick={(e) => handleClick("Madan", e)}>  
      Click Me</button>  
  );  
}
```

To include both a custom argument and the event object, use an arrow function that accepts custom arguments and event as input.



What are React Hooks ?

React Hooks are special functions that let you "hook into" React features from function components. Before Hooks were introduced in React 16.8 (in 2019), you had to use class components to access state and lifecycle features. Hooks changed that by bringing these powers to function components.



Why Hooks Matter

Think of Hooks like special tools that give your components superpowers. Without these tools, your function components could only receive data (props) and render content. They couldn't:

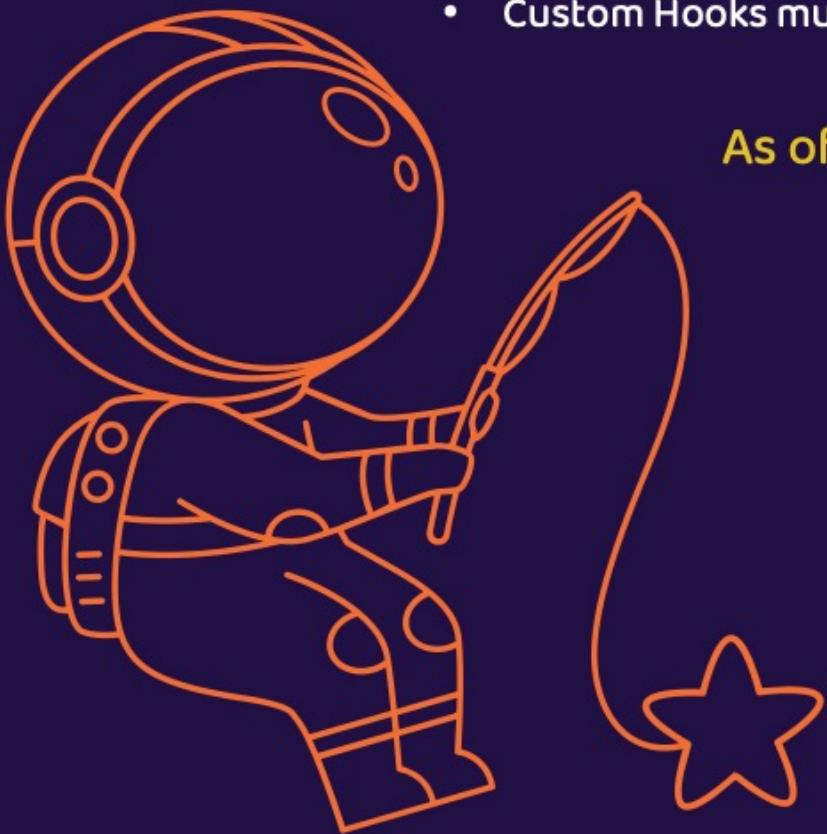
- Remember things between renders (like form input values)
- React to changes (like when data loads or when user clicked on a button)

Hooks solve these problems by letting function components have memory and awareness.

You can either use the built-in Hooks or combine them to build your own.

Rules to follow while using React Hooks

- Hooks can only be used inside React functional components.
- Hooks must be called at the top level of a component.
- Hooks cannot be used inside loops, conditions, or nested functions.
- Custom Hooks must start with use (e.g., useCustomHook).



As of the latest React version, there are 12 built-in Hooks in the React library.

- useState
- useReducer
- useContext
- useEffect
- useRef
- useImperativeHandle
- useLayoutEffect
- useInsertionEffect
- useMemo
- useCallback
- useTransition
- useDeferredValue

useState Hook

useState is a React Hook that lets you add a state variable to your component. React is going to track the changes on the state variable and re render the component based on the new state (data). useState provides a state variable and a function to update it. Below is the syntax

const [state, setState] = useState(initialValue);

- **state** → Holds the current value.
- **setState(newValue)** → Updates the value.
- **initialValue** → The starting value of the state.

Basic Example of useState

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
```



useMemo is a React Hook that lets you cache the result of a calculation between re-renders.. We can cache expensive calculations to avoid unnecessary re-executions. Helps optimize rendering in large applications. Below is the syntax of useMemo,

```
const cachedValue = useMemo(calculateValue, dependencies)
```

- **calculateValue** → The function calculating the value that you want to cache.
- **dependencies** -> The list of all reactive values referenced inside of the calculateValue code

On the initial render, useMemo returns the result of calling calculateValue with no arguments.

During next renders, it will either return an already stored value from the last render (if the dependencies haven't changed), or call calculateValue again, and return the result that calculateValue has returned.



Example usage

```
import { useMemo } from 'react';

function TodoList({ todos, tab, theme }) {
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
}
```

You need to pass two things to useMemo:

- A **calculation function** that takes no arguments, like `() =>`, and returns what you wanted to calculate.
- A **list of dependencies** including every value within your component that's used inside your calculation.
- On the initial render, the **value** you'll get from useMemo will be the result of calling your calculation.

On every subsequent render, React will compare the **dependencies** with the dependencies you passed during the last render. If none of the dependencies have changed (compared with `Object.is`), useMemo will return the value you already calculated before. Otherwise, React will re-run your calculation and return the new value.

In other words, useMemo caches a calculation result between re-renders until its dependencies change.

The `useEffect` Hook is a built-in React Hook that allows functional components to run code when the component renders or updates. `useEffect` runs after the component renders. That's why it is commonly used to synchronize a component with an external system. Below is the syntax,

`useEffect(setup, dependencies?)`

- **setup** → The function with your Effect's logic. Your setup function may also optionally return a cleanup function.
- **dependencies** → Optional. Dependencies controls when the effect runs

The Dependency Array in `useEffect`

Dependency Array	Behaviour
<code>useEffect(() => {})</code>	Runs on every render
<code>useEffect(() => {}, [])</code>	Runs only once when initial mounting is happening
<code>useEffect(() => {}, [count])</code>	Runs when count changes



useEffect Hook

Example usage

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] =
  useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
}
```

You need to pass two arguments to useEffect:

A **setup function** with setup code that connects to that system. It can return a optional **cleanup function** with cleanup code.

A **list of dependencies** including every value from your component used inside of those functions.

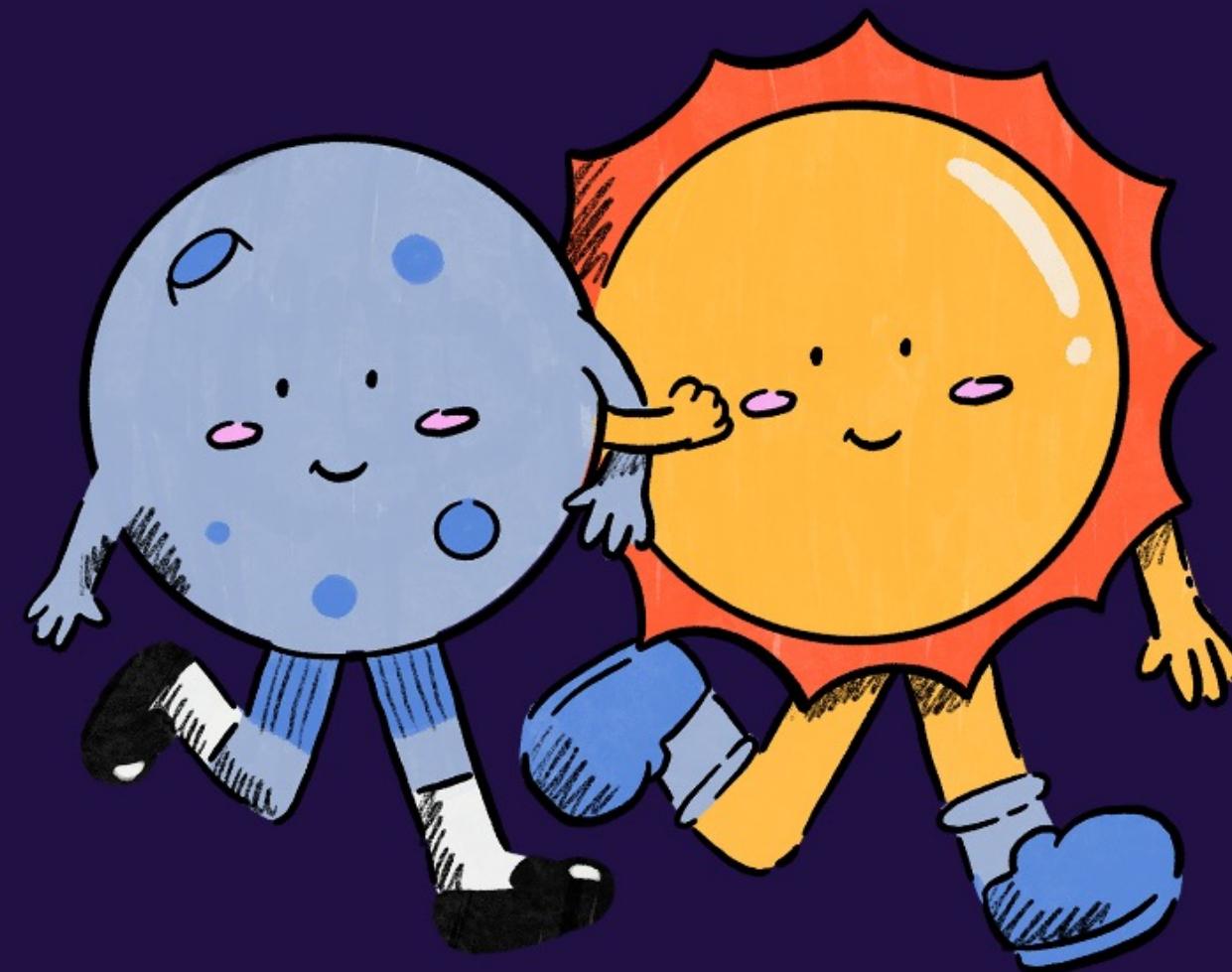
React calls your setup and cleanup functions whenever it's necessary, which may happen multiple times:

- 1.Your setup code runs when your component is added to the page (*mounts*).
- 2.After every re-render of your component where the dependencies have changed:
 - First, your cleanup code runs with the old props and state.
 - Then, your setup code runs with the new props and state.
- 3.Your cleanup code runs one final time after your component is removed from the page (*unmounts*).

Dark mode using Tailwind

Offering a dark version for your website is more of an expected feature rather than a nice to have nowadays.

To make this easier, Tailwind provides a dark mode variant, allowing you to customize your site's appearance when dark mode is active.



Enable Dark mode

All you need to do to enable dark mode for your Tailwind CSS project is to add the following code inside your main css file and then add the dark class on your html element.

index.css

```
@custom-variant dark (&:where(.dark, .dark *));
```

Explanation:

@custom-variant dark: Declares a new variant named dark.

(&:where(.dark, .dark *)): Specifies that the variant applies to elements with the dark class or any of their descendants.

Dark mode using Tailwind

Apply Dark Mode Styles Using the Variant

With the dark variant defined, you can now use it to style your components for dark mode:

```
<body class="bg-normalbg dark:bg-darkbg">  
  <!-- Content here -->  
</body>
```

Behind the scenes the code will look like below,

```
/* Normal (Light Mode) */  
.bg-normalbg {  
  background-color: #f3f4f6;  
}  
  
/* Dark Mode */  
.dark .bg-darkbg {  
  background-color: #1e2939;  
}
```

```
<html class="dark> // Need to add/remove this class  
<head>  
</head>  
<body class="bg-normalbg dark:bg-darkbg">  
  <div>Hello, World!</div>  
</body>  
</html>
```

The CSS selector `.dark.bg-darkbg` has higher specificity than `.bg-normalbg` because it has an additional parent selector (`.dark`). When two conflicting styles exist, the more specific selector wins.

Dark mode using Tailwind

To allow users to switch between light and dark modes, you can add a toggle function in the UI application which will add the dark class to the HTML element and to retain the theme values even after browser refresh/restart, store the theme details inside localStorage of the browser and load it in useEffect() hook

```
const [theme, setTheme] = useState(() => {
  return localStorage.getItem("theme") === "dark" ? "dark" : "light");
}

const toggleTheme = () => {
  setTheme((prevTheme) => {
    const newTheme = prevTheme === "light" ? "dark" : "light";
    localStorage.setItem("theme", newTheme);
    return newTheme;
  });
};

useEffect(() => {
  if (theme === "dark") {
    document.documentElement.classList.add("dark");
  } else {
    document.documentElement.classList.remove("dark");
  }
}, [theme]);
```

Introduction to Routing in React

Routing in React allows us to create multiple pages/views within a single-page application (SPA). It helps users navigate between different sections without reloading the page.

In traditional websites, clicking a link reloads the whole page. But in React apps, routing helps change only the necessary parts of the UI, making navigation faster and smoother.



Example of Routing in Websites

Think of a website like Amazon:

- `/` → Shows homepage
- `/products` → Shows product listings
- `/cart` → Shows shopping cart
- `/contact` → Shows contact form

With routing, you can click "Products", and the page updates without reloading.

Setting Up Routing in a React Project

React doesn't handle routing by default. We use a library called **react-router-dom** to manage navigation.



Step 1: Install React Router

```
npm install react-router-dom
```

Step 2: Create Pages

Create simple page components like `Home.jsx`, `About.jsx`, `Cart.jsx` and `Contact.jsx`

Step 3: Define Routes

Define route mappings with details on which component should be shown for a given URL path

Pass route mappings to `RouterProvider` and wrap the entire app with it

Defining Routes

```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <App />,
    errorElement: <ErrorPage />,
    children: [
      {
        index: true,
        element: <Home />,
      },
      {
        path: "/home",
        element: <Home />,
      },
      {
        path: "/about",
        element: <About />,
      },
      {
        path: "/contact",
        element: <Contact />,
      },
    ],
  },
]);
```

We need to use `createBrowserRouter()` from `react-router-dom` to define our App's route structure.

The Root Route (/)

```
{
  path: "/",
  element: <App />,
  errorElement: <ErrorPage />,
  children: [...]
```

`path: "/"` → Defines the root URL (default page).

`element: <App />` → This means the `<App />` component is the main layout.

`errorElement: <ErrorPage />` → If the user visits a wrong URL, React shows `<ErrorPage />`.

`children: [...]` → Inside the root layout, we have nested child routes.

Defining Routes

```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <App />,
    errorElement: <ErrorPage />,
    children: [
      {
        index: true,
        element: <Home />,
      },
      {
        path: "/home",
        element: <Home />,
      },
      {
        path: "/about",
        element: <About />,
      },
      {
        path: "/contact",
        element: <Contact />,
      },
    ],
  },
]);
```

The child routes (Nested inside /)

These routes define sub-pages of the application inside <App />.

Default Home Page (/)

```
{
  index: true,
  element: <Home />,
}
```

index: true → This means when a user visits / (root URL), it shows <Home />.

Index routes render into their parent's <Outlet/> at their parent's URL

Defining Routes

```
createRoot(document.getElementById("root")).render(  
  <StrictMode>  
    <RouterProvider  
      router={appRouter} />  
    </StrictMode>  
)
```

<RouterProvider router={appRouter} /> → Connects React Router to our application and loads all defined routes.

```
function App() {  
  
  return (  
    <>  
      <Header />  
      <Outlet />  
      <Footer />  
    </>  
  );  
}
```

The App component is the main layout of the application which is responsible to display Header, Outlet and Footer.

What is <Outlet />?

Think of <Outlet /> as a placeholder. It replaces itself with the correct page component depending on the current route.

Example:

- If the user visits /home, <Outlet /> shows the <Home /> page.
- If the user visits /about, <Outlet /> shows the <About /> page.

Defining Routes using Route

```
const routeDefinitions = createRoutesFromElements(  
  
  <Route path="/" element={<App />} errorElement={<ErrorPage />}>  
    <Route index element={<Home />} loader={productsLoader} />  
    <Route path="/home" element={<Home />} loader={productsLoader} />  
    <Route path="/about" element={<About />} />  
    <Route path="/contact" element={<Contact />} action={contactAction} />  
    <Route path="/login" element={<Login />} />  
    <Route path="/cart" element={<Cart />} />  
  </Route>  
);  
  
const appRouter = createBrowserRouter(routeDefinitions);
```

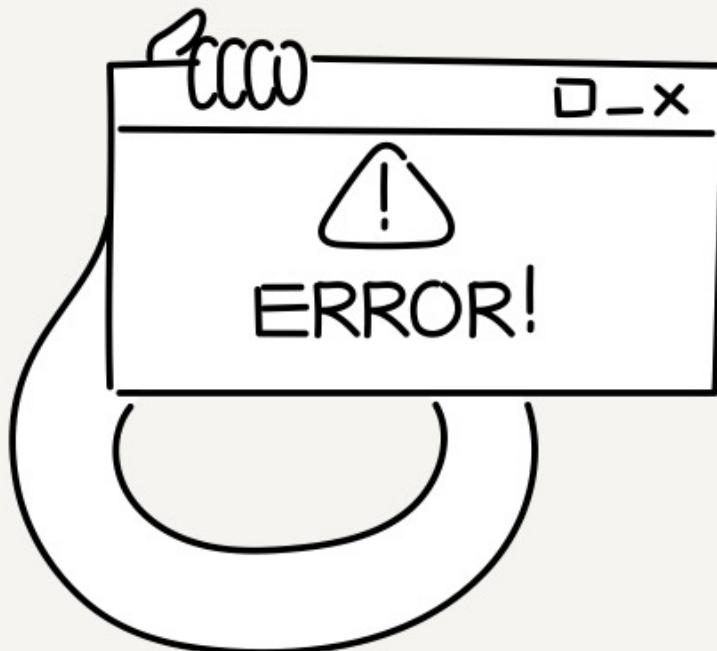
useRouteError()

useRouteError() is a hook from react-router-dom that:

- Catches errors that occur in loaders, actions, or route elements
- Prevents the app from crashing due to unexpected issues
- Displays a custom error message to users

```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <App />,
    errorElement: <ErrorPage />,
    children: [....],
  }
])
```

When an error happens in a route (e.g., failed API call), the user will be forwarded to ErrorPage component. Inside the same, we can show error details using **useRouteError()**.



```
import { useRouteError } from "react-router-dom";

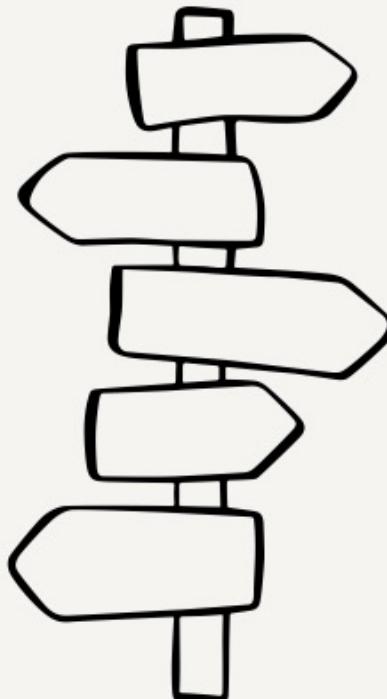
export default function ErrorPage() {
  const error = useRouteError();

  return (
    <div>
      <h1>Oops! Something went wrong. 😞</h1>
      <p>{error.statusText || error.message}</p>
    </div>
  );
}
```

In normal HTML, we use `` to navigate between pages.

But in React, we use `<Link>` instead of `<a>`, because React does not reload the page when using `<Link>`, making navigation faster and smoother.

How to Use `<Link>`



```
import { Link } from "react-router-dom";

function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/contact">Contact</Link>
    </nav>
  );
}
```

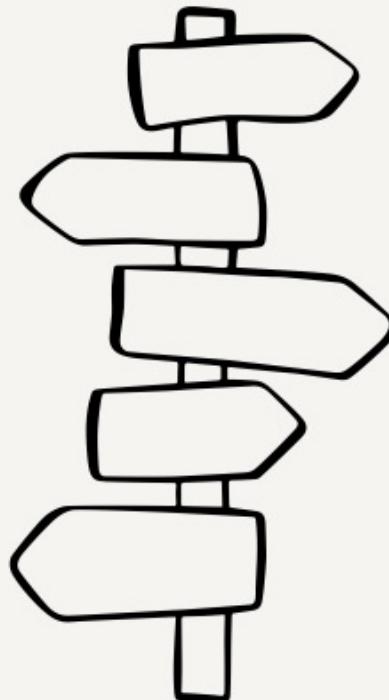
Clicking `<Link to="/">Home</Link>` will take the user to the Home page (/).

Clicking `<Link to="/about">About</Link>` will take the user to the About page (/about).

The page does not reload, making navigation instant.

<NavLink> is a special type of link in React Router that:

- Works like <Link> but automatically adds an "active" class to the currently selected link.
- Helps style the active page differently (e.g., bold, underline, change color).



```
import {NavLink} from "react-router-dom";

<NavLink to="/about"
  activeClassName={({ isActive }) =>
    isActive ? `underline ${navLinkClass}` : navLinkClass
  }>
  About
</NavLink>
```

isActive is true when the link matches the current page.

Use <Link> when you just need navigation. Use <NavLink> when you want to style the active page.

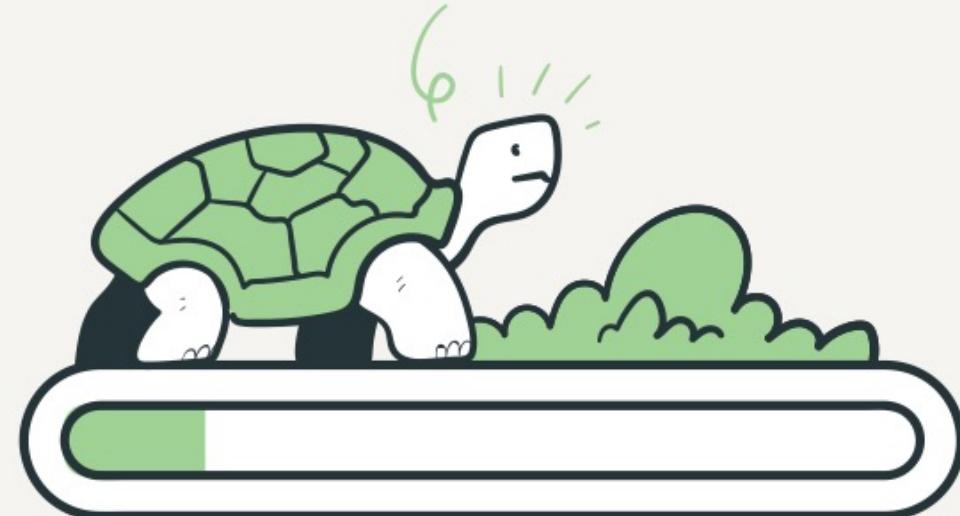
loader in React Router Dom

In React Router, **loaders** are used to fetch data from API or DB before rendering a component. They help improve performance and ensure that the page has all the necessary data before it loads.

Example: Using a loader to Fetch Data Before Rendering

First we define a function that fetches products

```
export async function productsLoader() {
  try {
    const response = await apiClient.get("/products");
    return response.data;
  } catch (error) {
    throw new Response(
      error.message, error.status);
  }
}
```



Now, we attach the loader to the route definition

```
const appRouter = createBrowserRouter([
  {
    path: "/home",
    element: <Home />,
    loader: productsLoader,
  },
]);
```

useLoaderData()

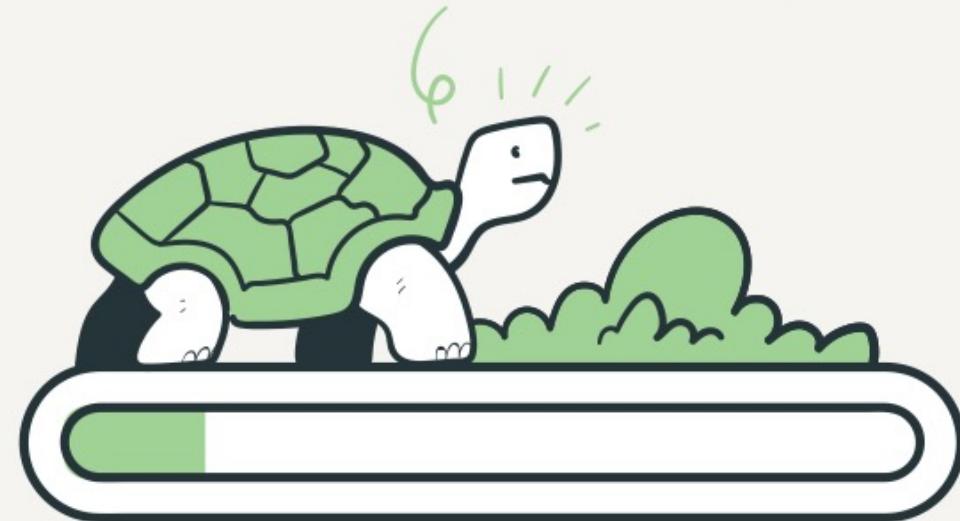
The `useLoaderData` hook will return the data that was fetched by the loader function from the closest route. Inside the component, we use `useLoaderData()` to get the fetched products.

```
import { useLoaderData } from "react-router-dom";

export default function Home() {

  const products = useLoaderData();

  return (
    <div>
      <h2>Products</h2>
      <ul>
        {products.map((product) => (
          <li key={product.id}>{product.title}</li>
        ))}
      </ul>
    </div>
  );
}
```



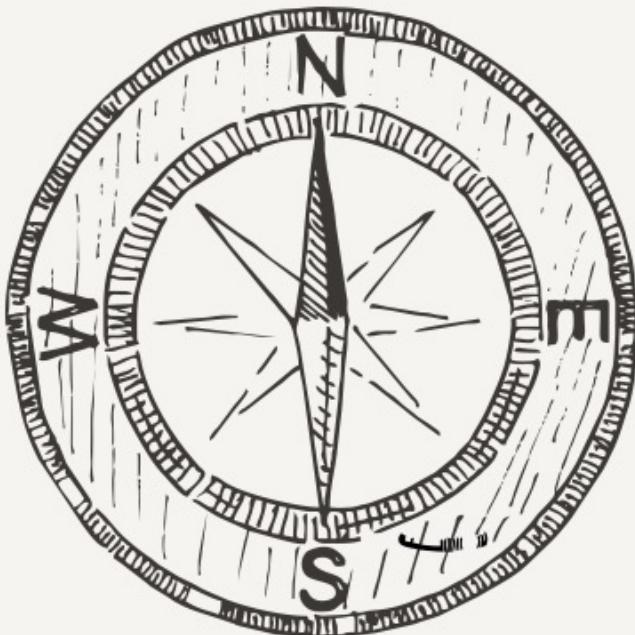
`useLoaderData()` automatically gets the data returned from `productsLoader()`. The page won't render until data is loaded.

useNavigation()

useNavigation() is a React Router hook that:

- Detects if a page is loading
- Helps show loading indicators (spinners, messages, etc.)
- Improves user experience by giving feedback during navigation

It is useful when navigating between pages, so users don't feel like the app is unresponsive. In the main layout, we check if a page is loading and show a loading message.



```
import { Outlet, useNavigation } from "react-router-dom";

function App() {
  const navigation = useNavigation();

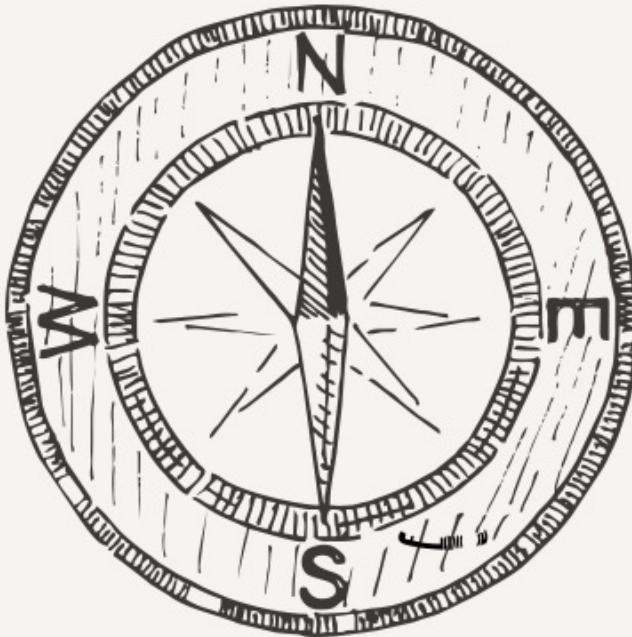
  return (
    <>
      <Header />
      {navigation.state === "loading" && <p>Loading...</p>}
      <Outlet />
      <Footer />
    </>
  );
}
```

useNavigate()

In React applications, we often need to navigate to a different page when an event like click button occurs. For this, we use programmatic navigation with `useNavigate()` from `react-router-dom`

What is `useNavigate()` ? - `useNavigate()` is a hook in React Router that allows us to change the page programmatically without using `<Link>`.

Let's say we have a Login Page where we want to navigate the user to the Dashboard Page after login.



```
import { useNavigate } from "react-router-dom";

function Login() {
  const navigate = useNavigate();
  const handleLogin = () => {
    navigate("/dashboard");
  };

  return (
    <div>
      <h2>Login Page</h2>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
}

export default Login;
```

`navigate(-1)` → Takes the user to the previous page.

`navigate(1)` → Takes the user to the next page.

Example: Navigating with Parameters : This will navigate to /product-details/123 and send product details as state.

```
function Product() {  
  const navigate = useNavigate();  
  
  const goToDetails = () => {  
    navigate("/product-details/123", { state: { name: "Laptop", price: 500 } });  
  };  
  
  return (  
    <div>  
      <h2>Product Page</h2>  
      <button onClick={goToDetails}>View Product Details</button>  
    </div>  
  );  
}
```

Replacing History Instead of Pushing - By default, `useNavigate()` adds a new entry to the browser history. If you want to replace the current history (so users can't go back), use `{ replace: true }`. This replaces the current page instead of adding a new history entry.

```
navigate("/dashboard", { replace: true });
```

useLocation()

The **useLocation** hook from react-router-dom provides access to the current URL's location object. It enables developers to track URL changes and conditionally render UI elements based on the path or search parameters.

If we pass data with `useNavigate()`, we can read it in the destination page using `useLocation()`.



```
import { useLocation } from "react-router-dom";

function ProductDetails() {
  const location = useLocation();
  console.log(location.pathname);
  const product = location.state;
  return (
    <div>
      <h2>Product Details</h2>
      <p>Name: {product.name}</p>
      <p>Price: ${product.price}</p>
    </div>
  );
}

export default ProductDetails;
```

action in React Router Dom

When building forms in React applications, we need to handle form submissions efficiently. With React Router we can build action functions that allow us to handle form data directly in routes instead of using onSubmit inside components.

Let's say we have a Contact Form, and we want to submit the form using React Router's action.

1) Create an action Function

```
export async function contactAction({ request, params }) {  
  const data = await request.formData();  
  
  const contactData = {  
    name: data.get("name"), email: data.get("email"),  
    message: data.get("message"),  
  };  
  
  try {  
    await apiClient.post("/contacts", contactData);  
    return { success: true };  
  } catch (error) {  
    throw new Response(  
      error.message, error.status);  
  }  
}
```



action in React Router Dom

2) Create the Contact Form Using <Form>

Instead of using <form> with onSubmit, we use React Router's <Form> component.

```
export default function Contact() {
  const actionData = useActionData();

  return (
    <div>
      <h2>Contact Us</h2>
      {actionData?.success && <p> Form submitted successfully!</p>}
      <Form method="post">
        <label>Name:</label>
        <input type="text" name="name" required />

        <label>Email:</label>
        <input type="email" name="email" required />

        <label>Message:</label>
        <textarea name="message" required></textarea>

        <button type="submit">Submit</button>
      </Form>
    </div>
  );
}
```



useActionData() hook provides the returned value from the previous navigation's action result

3) Define the Route with an action

Now, we attach the action function to the route.

```
const appRouter = createBrowserRouter([
  {
    path: "/contact",
    element: <Contact />,
    action: contactAction,
  },
]);
```



4) Redirecting After Submission

If we want to redirect the user after form submission (e.g., to a Home page), we use `redirect()`.

```
import { redirect } from "react-router-dom";

export async function contactAction({ request }) {
  const formData = await request.formData();
  .....
  return redirect("/home");
}
```

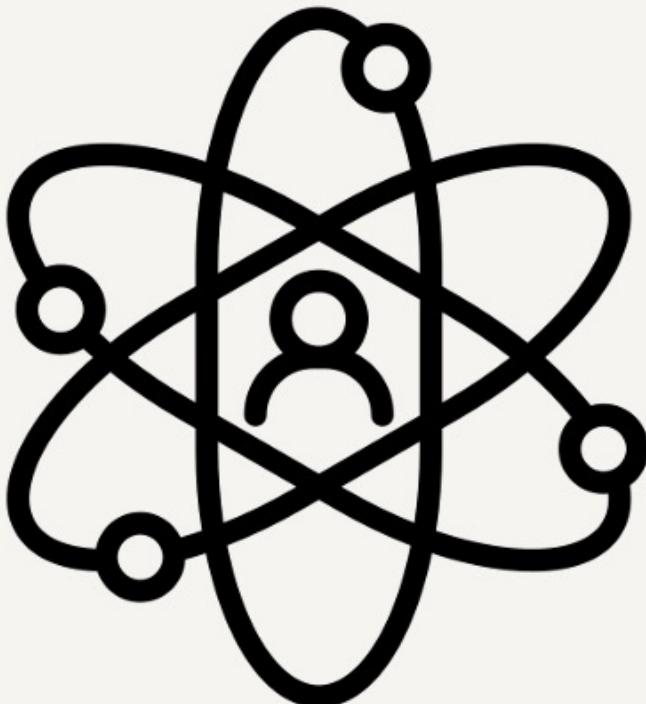
`useSubmit()` hook can be used to invoke the action without using `<Form>`. It lets you submit a form manually from a function

Building Dynamic Routes

Dynamic routes are routes that change based on a parameter

Example:

- Static Route: `/about`
- Dynamic Route: `/products/:productId`



```
<Link to={`/products/${product.productId}`} state={{ product }} />
<Route path="/products/:productId" element={<ProductDetail />} />
```

The `:productId` in `/products/:productId` is a dynamic parameter. By using **useParams()** Hook inside component we can extract the productId from the URL.

Any additional data can be sent using state property and the same can be accessed in the resultant component using **useLocation()**

```
const param = useParams();
const productId = param.productId;

const location = useLocation();
const product = location.state?.product;
```

Spring Boot DevTools is a developer tool that improves the development experience by:

- ✓ Auto-restarting the application when code changes.
- ✓ Disabling caching in development mode so changes take effect immediately.

To enable DevTools, add the dependency in your pom.xml,

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
```

Spring Boot automatically disables DevTools in production for security and performance. You don't need to remove it—it runs only in development mode.



Spring Boot Actuator

Spring Boot Actuator is a tool that helps monitor and manage a Spring Boot application. It provides ready-made REST APIs to check the health, metrics, and status of your app.

With Actuator, you can:

- Check if the app is running properly.
- Get system metrics (memory, CPU, threads).
- Monitor HTTP requests, database connections.
- See which Spring Beans are loaded.

To enable Actuator, add this dependency in pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```



Spring Boot Actuator

By default, only the health check (/actuator/health) endpoint is enabled.

To enable more endpoints, update application.properties:

```
management.endpoints.web.exposure.include=*
```

You can add custom information in application.properties:

```
management.info.env.enabled=true
info.app.name=eazystore
info.app.description=Eazy Store
info.app.version=1.0.0
```

Now, if you visit /actuator/info, you will see:

```
{
  "app": {
    "name": "eazystore",
    "description": "Eazy Store",
    "version": "1.0.0"  }
}
```



Spring Boot Actuator

To check Application Health -> Access the path <http://localhost:8080/actuator/health> to see below response,

```
{  
  "status": "UP"  
}
```

To view Metrics -> Access the path
<http://localhost:8080/actuator/metrics> to see response like below,

```
{  
  "names": [  
    "jvm.memory.used",  
    "system.cpu.usage",  
    "http.server.requests",  
    "logback.events"  
  ]  
}
```



To get specific metric details like CPU usage, visit: <http://localhost:8080/actuator/metrics/system.cpu.usage>

Common Actuator Endpoints

Endpoint	What It Does
/actuator/health	Checks if the app is running
/actuator/info	Shows custom app information
/actuator/metrics	Displays memory, CPU, request stats
/actuator/env	Shows environment variables
/actuator/beans	Lists all Spring Beans in the app
/actuator/mappings	Displays all API endpoints
/actuator/loggers	Shows logs and log levels
/actuator/threaddump	Shows running threads

Spring Boot Actuator

Is Actuator Safe? Yes! But you should secure it in production.

By default:

Sensitive data is protected.

Only health and info are public.

You can restrict access using Spring Security.



In a Spring Boot REST API, handling requests and responses efficiently is crucial for building robust applications. Spring Boot provides several annotations, classes, and interfaces to process incoming requests and return appropriate responses.

Using @RequestBody to Accept JSON Data

When a client sends JSON data in the request body, we use `@RequestBody` to bind it to a Java object. Spring Boot automatically converts JSON to Java objects using Jackson (default library for JSON processing).

```
@RestController
@RequestMapping("/api/v1/dummy")
public class DummyController {

    @PostMapping("/create")
    public ResponseEntity<String> createUser(@RequestBody UserDto user) {
        return ResponseEntity.ok("User created successfully");
    }
}
```



Using @RequestParam to Accept Query Parameters

Use `@RequestParam` when parameters are sent as part of the URL.

Example Request: `GET /api/v1/dummy/search?name=John`

```
@GetMapping("/search")
public ResponseEntity<String> searchUser(@RequestParam String name) {
    return ResponseEntity.ok("Searching for user: " + name);
}
```

Making @RequestParam Optional

By default, `@RequestParam` is required. If you do not pass the parameter, the server will return a 400 Bad Request error. To make it optional, set `required = false`:

```
@GetMapping("/user")
public String getUser(@RequestParam(required = false) String name) {
    return "Hello, " + (name != null ? name : "Guest");
}
```

@RequestParam

Setting Default Values

You can provide a default value to avoid null when the parameter is missing.

```
@GetMapping("/user")
public String getUser(@RequestParam(defaultValue = "Guest") String name) {
    return "Hello, " + name;
}
```

Custom Parameter Names

By default, the method argument name is used as the parameter name, but you can customize it.

```
@GetMapping("/user")
public String getUser(@RequestParam("user_name") String name) {
    return "Hello, " + name;
}
```

Multiple Parameters

You can accept multiple request parameters

```
@GetMapping("/greet")
public String greet(@RequestParam String firstName, @RequestParam String lastName) {
    return "Hello, " + firstName + " " + lastName;
}
```

Using @RequestParam with Map

You can retrieve all request parameters dynamically using a Map<String, String>.

```
@GetMapping("/params")
public String getParams(@RequestParam Map<String, String> params) {
    return "Parameters: " + params;
}
```

Using @PathVariable to Accept Path Parameters

@PathVariable is used when parameters are part of the URL path. Example Request: GET /api/v1/dummy/search/10

```
@GetMapping("/search/{id}")
public ResponseEntity<String> getUserId(@PathVariable Long id) {
    return ResponseEntity.ok("User ID: " + id);
}
```

All the concepts of @RequestParam are also applicable and same for @PathVariable as well

Using @RequestHeader to Accept Headers

Sometimes, we need to extract custom headers from the request.

```
@GetMapping("/headers")
public ResponseEntity<String> getHeader(@RequestHeader("User-Agent") String userAgent) {
    return ResponseEntity.ok("User-Agent: " + userAgent);
}
```

Using Map<String, String> to accept all headers

```
@GetMapping("/headers")
public ResponseEntity<String> getAllHeaders(@RequestHeader Map<String, String> headers) {
    return ResponseEntity.ok("Received Headers: " + headers.toString());
}
```

Using HttpHeaders for additional flexibility

```
@GetMapping("/headers")
public ResponseEntity<String> getHttpHeaders(@RequestHeader HttpHeaders headers) {
    return ResponseEntity.ok("Received Headers: " + headers);
}
```

RequestEntity & ResponseEntity

Using RequestEntity to Access Full HTTP Request

RequestEntity allows accessing the complete request, including headers and body

```
@PostMapping("/request-entity")
public ResponseEntity<String> handleRequestEntity(RequestEntity<UserDto> requestEntity) {
    HttpHeaders headers = requestEntity.getHeaders();
    UserDto user = requestEntity.getBody();
    return ResponseEntity.ok("Received user: " + user.getName() + ", Headers: " + headers);
}
```

Using ResponseEntity for Custom Responses

ResponseEntity allows customizing the HTTP status, headers, and response body.

```
@GetMapping("/response")
public ResponseEntity<String> customResponse() {
    return ResponseEntity.status(HttpStatus.CREATED)
        .header("Custom-Header", "ExampleValue")
        .body("User created successfully");
}
```

Global Exception Handling in Spring Boot

In a Spring Boot application, when an exception occurs, it may cause an application crash or return an unclear error response to the client. **Global Exception Handling** ensures that errors are handled consistently and meaningful error messages are returned.

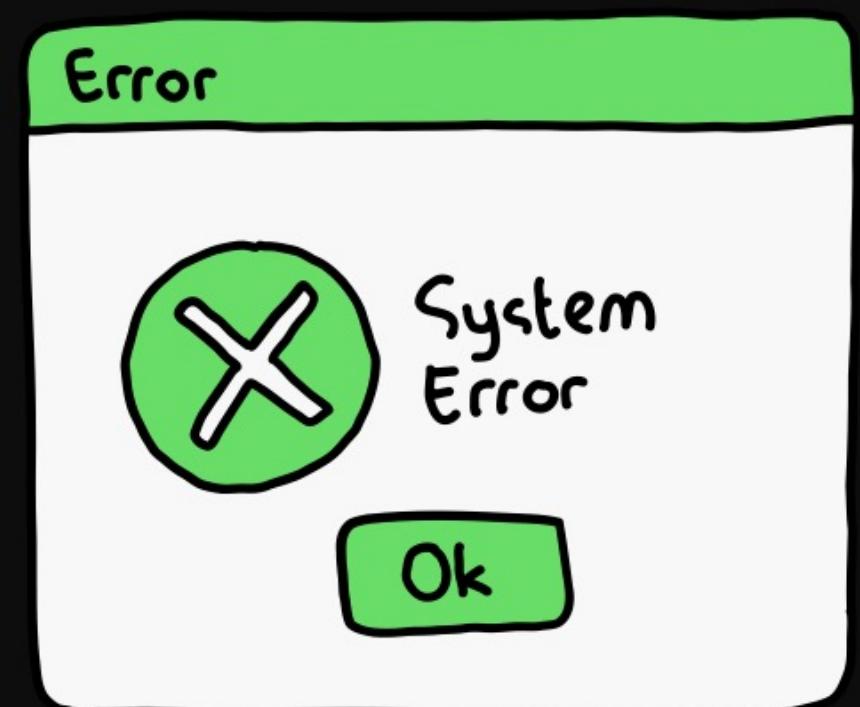
Why Use Global Exception Handling?

- ✓ Provides consistent error responses across the application.
- ✓ Improves readability and maintainability of exception handling logic.
- ✓ Helps in logging and debugging by capturing errors in a central place.
- ✓ Ensures better user experience with meaningful error messages.
- ✓ No need to write try and catch blocks in all the methods

Ways to Handle Exceptions in Spring Boot

Spring Boot provides two main ways to handle exceptions globally:

- ✓ Using `@ExceptionHandler` (Controller-Level Handling)
- ✓ Using `@RestControllerAdvice & @ExceptionHandler` (Global Exception Handling)



Documenting REST APIs is essential for making them understandable and usable by other developers.

[springdoc-openapi](#) is a Java library that automatically generates API documentation for your Spring Boot REST APIs. It follows the OpenAPI (Swagger) standard.

Why Document REST APIs?

- ✓ Helps developers understand available endpoints and their request/response formats.
- ✓ Reduces dependency on manual explanations.
- ✓ Improves API usability and onboarding.
- ✓ Makes integration with other systems easier.

Developers can further enhance the default API documentation using annotations like `@Schema`, `@Tag`, `@Operation`, `@ApiResponse` etc.



To use the Springdoc library, Developer need to add the below dependency in pom.xml,

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.8.5</version>
</dependency>
```

Once the dependency added, Start your Spring Boot application, and open:

🔗 Swagger UI:

👉 <http://localhost:8080/swagger-ui.html>

🔗 OpenAPI JSON:

👉 <http://localhost:8080/v3/api-docs>

🔥 Spring automatically generates the API documentation!



For more details, refer <https://springdoc.org/>

Validations in Spring Boot REST APIs

Validation ensures that the data sent in API requests is correct before processing.

Why Use Validation?

- Ensures correct and expected data input.
- Prevents invalid data from reaching the database.
- Reduces errors and improves API reliability.
- Helps maintain clean and secure APIs.

To get started with validation, add the below dependency,

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Spring Boot provides two key annotations to trigger validation:

- @Valid** – Used on method parameters to enable validation.
- @Validated** – Used at the class level for group-based validation.



Validations in Spring Boot REST APIs

Spring Boot supports field-level validation using Jakarta Validation annotations inside DTOs and Entity classes

```
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Pattern;
import jakarta.validation.constraints.Size;

public class ContactRequestDto {

    @NotBlank(message = "Name cannot be blank")
    @Size(min = 5, max = 30, message = "Name must be between 5 and 30 characters")
    private String name;

    @Email(message = "Invalid email format")
    @NotBlank(message = "Email cannot be blank")
    private String email;

    @Pattern(regexp = "^\d{10}$", message = "Mobile number must be exactly 10 digits")
    private String mobileNumber;

    @NotBlank(message = "Message cannot be empty")
    @Size(min = 5, max = 500, message = "Message must be between 5 and 500 characters")
    private String message;
}
```

Validations in Spring Boot REST APIs

To apply validation on RequestBody data, use `@Valid` inside a controller method.

```
@RestController
@RequestMapping("api/v1/contacts")
@RequiredArgsConstructor
public class ContactController {

    private final IContactService iContactService;

    @PostMapping
    public ResponseEntity<String> saveContact(@Valid @RequestBody ContactRequestDto contactRequestDto)
    {
        boolean isSaved = iContactService.saveContact(contactRequestDto);
        return ResponseEntity.status(HttpStatus.CREATED)
            .body("Request processed successfully");
    }
}
```

Validations in Spring Boot REST APIs

To apply validation on RequestParam, use @Validated on top of the controller class

```
@RestController
@RequestMapping("api/v1/dummy")
@Validated
public class DummyController {

    @GetMapping("/search")
    public String searchUser(@Size(min = 5, max = 30) @RequestParam(required = false, defaultValue =
"Guest",
            name = "name") String userName) {
        return "Searching for user : " + userName;
    }

}
```

Best Practices For Validations

- Use DTOs instead of entity classes for request validation.
- Always return meaningful error messages to the client.
- Use @Valid for method-level validation and @Validated for class-level validation.
- Centralize validation error handling using @RestControllerAdvice.

Validations in Spring Boot REST APIs

Global Exception Handling for Validation Errors

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>> handleValidationExceptions(
        MethodArgumentNotValidException exception) {
        Map<String, String> errors = new HashMap<>();
        List<FieldError> fieldErrorList = exception.getBindingResult().getFieldErrors();
        fieldErrorList.forEach(error -> errors.put(error.getField(), error.getDefaultMessage()));
        return ResponseEntity.badRequest().body(errors);
    }

    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<Map<String, String>> handleConstraintViolationException(
        ConstraintViolationException exception) {
        Map<String, String> errors = new HashMap<>();
        Set<ConstraintViolation<?>> constraintViolationSet = exception.getConstraintViolations();
        constraintViolationSet.forEach(constraintViolation ->
            errors.put(constraintViolation.getPropertyPath().toString(),
                constraintViolation.getMessage()));
        return ResponseEntity.badRequest().body(errors);
    }
}
```

Auditing in Spring Boot with JPA

Auditing in JPA helps track changes to entities by automatically recording who created or updated a record and when the changes occurred.

Why Use Auditing?

- Helps in maintaining logs of data modifications.
- Useful for tracking user actions in the database.
- Reduces manual effort by automatically capturing timestamps and user details.
- Ensures better data integrity and accountability.

Enable JPA Auditing

To enable auditing, we need to:

1. Create a `BaseEntity` class to handle auditing fields.
2. Use Spring Security or a custom implementation to capture the user who modified the data.
3. Enable Auditing in Spring Boot using `@EnableJpaAuditing`.



Auditing in Spring Boot with JPA

Step 1: Create a BaseEntity Class

```
@Getter  
@Setter  
@MappedSuperclass  
@EntityListeners(AuditingEntityListener.class)  
public class BaseEntity {  
  
    @CreatedDate  
    @Column(name = "created_at", nullable = false, updatable = false)  
    @CreationTimestamp  
    private Instant createdAt;  
  
    @CreatedBy  
    @Column(name = "created_by", nullable = false, length = 20, updatable = false)  
    private String createdBy;  
  
    @LastModifiedDate  
    @Column(name = "updated_at", insertable = false)  
    @UpdateTimestamp  
    private Instant updatedAt;  
  
    @LastModifiedBy  
    @Column(name = "updated_by", length = 20, insertable = false)  
    private String updatedBy;  
}
```

Auditing in Spring Boot with JPA

Step 2: Modify the Entity classes to extend BaseEntity

```
@Getter  
@Setter  
@Entity  
@Table(name = "contacts")  
public class Contact extends BaseEntity {  
  
    // other fields  
  
}
```

Step 3: Create an AuditorAware Implementation

We need to tell Spring who the current user is when saving/updating entities. If you use Spring Security, modify the `getCurrentAuditor()` method to return the authenticated username.

```
@Component("auditorAwareImpl")  
public class AuditorAwareImpl implements AuditorAware<String> {  
  
    @Override  
    public Optional<String> getCurrentAuditor() {  
        return Optional.of("Anonymous user");  
    }  
}
```

Auditing in Spring Boot with JPA

Step 4: Enable Auditing in the Main Class

```
@SpringBootApplication
@EnableJpaAuditing(auditorAwareRef = "auditorAwareImpl")
public class EazystoreApplication {

    public static void main(String[] args) {
        SpringApplication.run(EazystoreApplication.class, args);
    }

}
```

Logging in Spring Boot

✓ What is Logging?

- Logging is the process of recording important events in an application.
- Helps in debugging, monitoring, and troubleshooting.
- Logs provide information like errors, warnings, execution flow, and performance insights.

✓ Key benefits of logging:

1. Debugging – Helps in finding and fixing errors.
2. Monitoring – Tracks application behavior over time.
3. Auditing – Keeps a record of important actions.
4. Performance Analysis – Helps in identifying slow parts of the app.

Spring Boot uses SLF4J (Simple Logging Facade for Java) with Logback by default. No need to add dependencies—it works out of the box.

Default log output example:

```
2044-03-20 10:15:30 INFO com.example.MyApp - Application  
started successfully!
```



Logging in Spring Boot

Formatting Logs (Pattern Configuration)

```
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
```

Example Output

```
2044-03-20 10:30:45 [main] INFO com.example.MyService - Application started!
```

Spring Boot supports different logging frameworks:

Logging Framework	Description
SLF4J (Facade)	Generic logging API (recommends best practice)
Logback (Default)	The default Spring Boot logger
Log4j2	Powerful and feature-rich logging
Java Util Logging (JUL)	Built-in Java logging



Logging in Spring Boot

Log Levels in Spring Boot. By default, Spring Boot sets the root log level to **INFO**

Log Level	Description	Example Use Case
TRACE	Most detailed logs	Tracking method calls
DEBUG	Debugging information	Variable values, steps in execution
INFO	General application info	Startup messages, business events
WARN	Potential issues	Deprecated API usage, high memory usage
ERROR	Serious problems	Exceptions, failed transactions

Impact of Different Log Levels

Configured Log Level	Logs That Appear
TRACE	Shows TRACE, DEBUG, INFO, WARN, ERROR
DEBUG	Shows DEBUG, INFO, WARN, ERROR
INFO	Shows INFO, WARN, ERROR
WARN	Shows WARN, ERROR
ERROR	Shows only ERROR

Logging in Spring Boot

Changing Log Levels in application.properties

Set log level for the entire application:

```
logging.level.root=DEBUG
```

Set log level for a specific package:

```
logging.level.com.example.myapp=DEBUG
```

By default, logs are printed to the console. To save them in a file, configure:

```
# Save logs to a file
logging.file.name=/Users/eazybytes/Desktop/logs/app.log
```

Using JSON Logs for Structured Logging

To enable JSON logs for easy parsing in ELK Stack or Cloud Logging:

```
logging.pattern.file={"timestamp":"%d{yyyy-MM-dd HH:mm:ss}","level":"%p","logger":"%c","message":"%m"}
```



Logging in Spring Boot

Logging example,

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
@RequestMapping("api/v1/logging")
public class LoggingController {

    private static final Logger log = LoggerFactory.getLogger(LoggingController.class);

    @GetMapping
    public ResponseEntity<String> testLogging() {
        log.trace("🔍 TRACE: This is a very detailed trace log. Used for tracking execution flow.");
        log.debug("🐞 DEBUG: This is a debug message. Used for debugging.");
        log.info("ℹ️ INFO: This is an informational message. Application events.");
        log.warn("⚠️ WARN: This is a warning! Something might go wrong.");
        log.error("❗️ ERROR: An error occurred! This needs immediate attention.");
        return ResponseEntity.status(HttpStatus.OK).
            body("Logging tested successfully");
    }
}
```

Logging in Spring Boot

Logging example with Lombok,

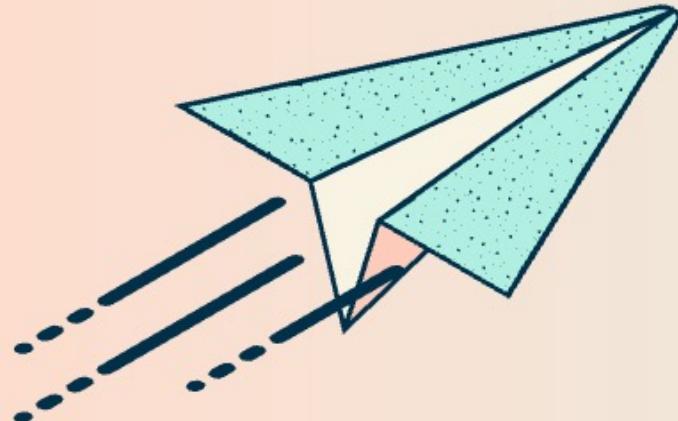
```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
@RequestMapping("api/v1/logging")
@Slf4j
public class LoggingController {

    @GetMapping
    public ResponseEntity<String> testLogging() {
        log.trace("🔍 TRACE: This is a very detailed trace log. Used for tracking execution flow.");
        log.debug("🐞 DEBUG: This is a debug message. Used for debugging.");
        log.info("ℹ️ INFO: This is an informational message. Application events.");
        log.warn("⚠️ WARN: This is a warning! Something might go wrong.");
        log.error("🔥 ERROR: An error occurred! This needs immediate attention.");
        return ResponseEntity.status(HttpStatus.OK).
            body("Logging tested successfully");
    }
}
```

Lombok `@Slf4j` automatically creates the logger variable `log` ! No need for `LoggerFactory`!

In Spring, a bean is an object that is managed by the Spring container. When you define a bean, you can also define its scope—which controls how and when Spring creates a new instance of that bean.



Spring provides five major bean scopes:

01

Singleton (Default)

02

Prototype

03

Request (for Web Apps)

04

Session (for Web Apps)

05

Application (for Web Apps)

Spring Bean Scopes - Request, Session, Application

Request Scope (@RequestScope)

What is it? - A new bean instance is created for each HTTP request.

When to use? - When a bean should be valid only for the duration of a request (e.g., storing request-specific data).

Use case - Storing temporary user inputs like form data

```
@Component  
@RequestScope  
@Getter @Setter  
public class RequestScopedBean {  
  
    private String username;  
}
```

How it works?

- Each HTTP request gets a new instance of RequestScopedBean.
- If multiple users send requests, each gets a separate instance.
- As soon as the request is complete, the bean is destroyed.



Spring Bean Scopes - Request, Session, Application

Session Scope (@SessionScope)

What is it? - A new bean instance is created for each user session.

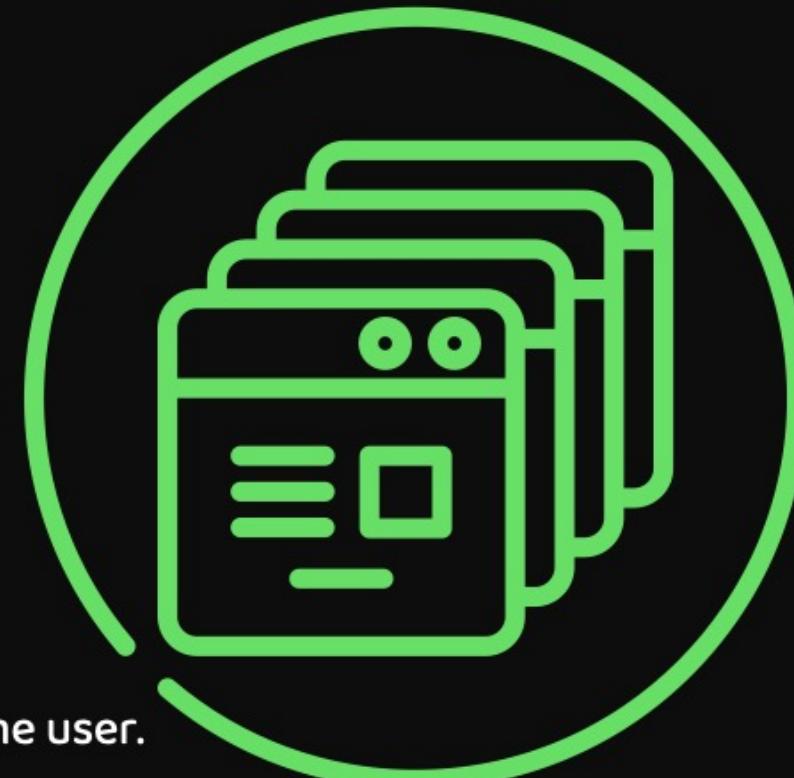
When to use? - When you need to store user-specific data across multiple requests in a single session.

Use case - Storing logged-in user details, shopping cart information.

```
@Component  
@SessionScope  
@Getter @Setter  
public class SessionScopedBean {  
  
    private String username;  
}
```

How it works?

- When a user logs in, a SessionScopedBean is created.
- The same bean instance is used for multiple requests from the same user.
- When the session expires, the bean is destroyed.



Spring Bean Scopes - Request, Session, Application

Application Scope (@ApplicationScope)

What is it? - A single bean instance is shared across the entire application.

When to use? - When you need global data shared among all users and requests.

Use case - Storing global statistics like visitor count

```
@Component
@ApplicationScope
@Getter
public class ApplicationScopedBean {
    private int visitorCount = 0;

    public void incrementVisitorCount() {
        visitorCount++;
    }
}
```

How it works?

- Only one instance of ApplicationScopedBean is created.
- It is shared across all users and requests.
- The bean is destroyed only when the application stops.



Spring Bean Scopes - Request, Session, Application

Using Scoped Beans in Controllers

```
@RestController @RequiredArgsConstructor
@RequestMapping("/api/v1/scope")
public class ScopeController {

    private final RequestScopedBean requestScopedBean;
    private final SessionScopedBean sessionScopedBean;
    private final ApplicationScopedBean applicationScopedBean;

    @GetMapping("/request")
    public String getRequestScope() {
        requestScopedBean.setUsername("JohnDoe");
        return "Request Scope: " + requestScopedBean.getUsername();
    }

    @GetMapping("/session")
    public String getSessionScope() {
        sessionScopedBean.setUsername("JohnDoe");
        return "Session Scope User: " + sessionScopedBean.getUsername();
    }

    @GetMapping("/application")
    public String getApplicationScope() {
        applicationScopedBean.incrementVisitorCount();
        return "Application Visitor Count: " + applicationScopedBean.getVisitorCount();
    }
}
```

Difference between @ApplicationScope and @Singleton scope

Use `@ApplicationScope` when working with web applications to store global web-related state.

Use Singleton (default scope) for services, repositories, or shared business logic in all types of Spring applications.

Singleton beans are managed by the Spring container, whereas Application-scoped beans are managed by the Servlet context.

Feature	<code>@ApplicationScope</code>	<code>@Singleton</code>
Scope	One instance per Spring web application	One instance per Spring container (default scope)
Usage	Works only in web applications (Spring MVC, Spring Boot Web)	Works in all Spring applications (Web, CLI, Batch, etc.)
Annotation	<code>@ApplicationScope</code>	No annotation required (default), but can use <code>@Scope("singleton")</code>
Instance Sharing	Shared across the entire application for all users and requests	Shared across the entire Spring container
Bean Destruction	Destroyed when the web application stops	Destroyed when the Spring container shuts down
Use Case	Storing global application state (e.g., visitor count, app-wide config)	Storing singleton service beans (e.g., <code>@Service</code> , <code>@Repository</code> , <code>@Component</code>)

What is Props Drilling in React?

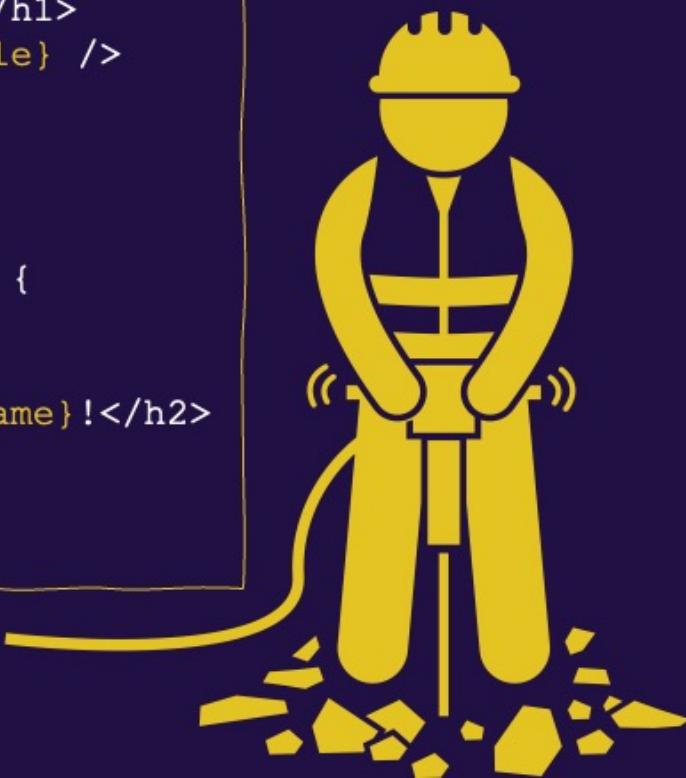
Props drilling refers to the practice of passing data (props) through multiple layers of components in a React application, even when intermediate components do not need the data. This happens when a deeply nested child component requires access to data from a higher-level parent component.

Example of Props Drilling

```
function App() {  
  const [profile, setProfile] = useState({  
    name: "Madan", role: "Instructor" });  
  
  return (  
    <div>  
      <Header profile={profile} />  
    </div>  
  );  
}
```

The **profile** prop is passed through Header, even though it don't use it. Such properties, we call them as **elongated prop**

```
function Header({ profile }) {  
  return (  
    <header>  
      <h1>This is the header</h1>  
      <Content profile={profile} />  
    </header>  
  );  
}  
  
function Content({ profile }) {  
  return (  
    <main>  
      <h2>Welcome, {profile.name}!</h2>  
    </main>  
  );  
}
```



What is Props Drilling in React?

What is elongated prop ?

An elongated prop is a prop that is not consumed but it is only passed down to another component. When a component receives a prop from its parent and doesn't consume the prop, it passes the prop down to another component. This prop is called elongated prop because it has been extended.

Problems with the props drilling

- Header doesn't need the profile prop but is forced to pass it down.
- If the component structure gets deeper, prop drilling becomes harder to manage.
- Leads to unnecessary re-renders.



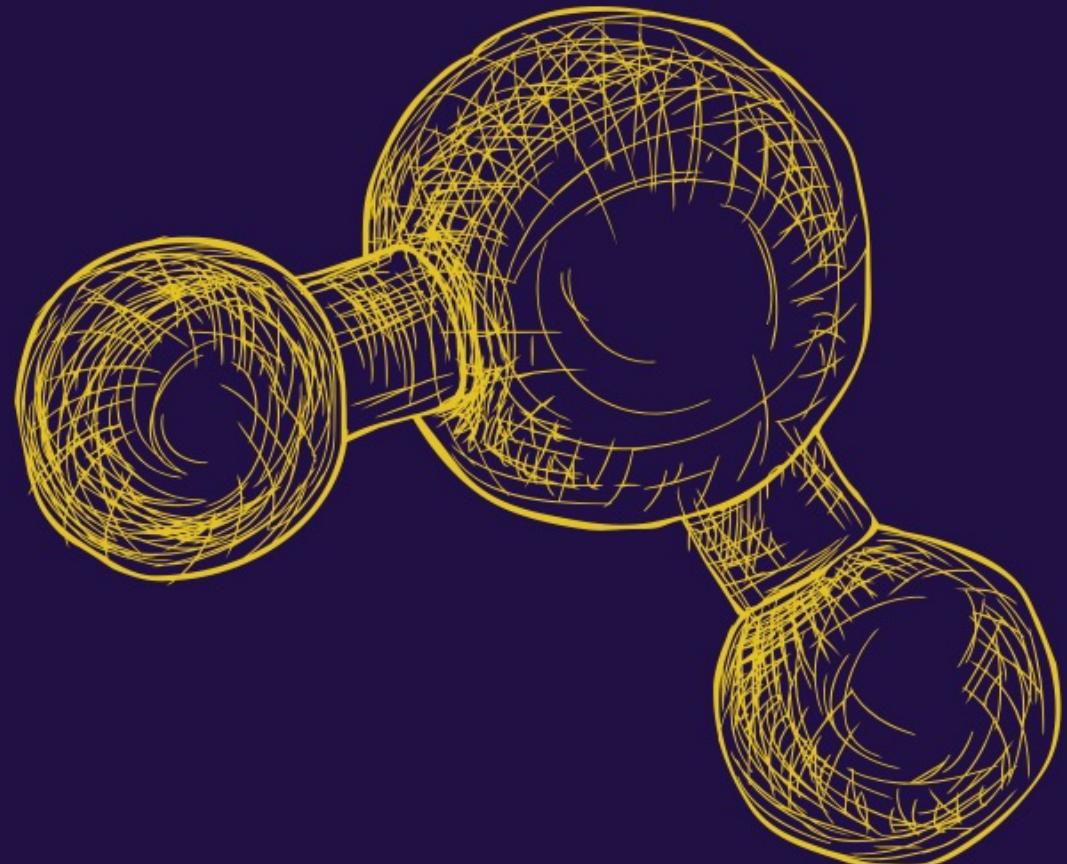
Avoiding Prop Drilling Using Component Composition

Component Composition is the practice of combining multiple React components to build more complex UIs.

Real-Life Example

💡 Think of React Components like LEGO blocks!

- Small LEGO pieces = Small components (Button, Input, Card)
- Combine pieces = Build bigger structures (Navbar, Sidebar)
- Final structure = A full UI (Dashboard, Page)



Avoiding Prop Drilling Using Component Composition

Fix 1: Instead of passing props manually, we pass components as children, keeping the data inside the parent (App) context.

```
function App() {  
  const [profile, setProfile] = useState({  
    name: "Madan", role: "Instructor"  
  });  
  
  return (  
    <div>  
      <Header>  
        <Content profile={profile} />  
      </Header>  
    </div>  
  );  
}
```

```
function Header({ children }) {  
  return (  
    <header>  
      <h1>This is the header</h1>  
      {children} /* Profile is now passed naturally */  
    </header>  
  );  
}  
  
function Content({ profile }) {  
  return (  
    <main>  
      <h2>Welcome, {profile.name}!</h2>  
    </main>  
  );  
}
```

Avoiding Prop Drilling Using Component Composition

Fix 2: Passing children Explicitly

```
function App() {  
  const [profile, setProfile] = useState({  
    name: "Madan", role: "Instructor" });  
  
  return (  
    <div>  
      <Header  
        children=  
          {<Content profile={profile} />}  
      />  
    </div>  
  );  
}
```

⌚ What We Fixed

- No unnecessary prop passing through Header.
- profile stays inside App, where it belongs.
- More readable and scalable code.

```
function Header({ children }) {  
  return (  
    <header>  
      <h1>This is the header</h1>  
      {children}  
    </header>  
  );  
}  
  
function Content({ profile }) {  
  return (  
    <main>  
      <h2>Welcome, {profile.name}!</h2>  
    </main>  
  );  
}
```

What is "Lifting State Up" in React?

When two components need to stay in sync, it's best to move their shared state to their closest common parent. This process is called "lifting state up."

Instead of keeping the state separately in each component, you store it in the parent and pass it down as props. This ensures both components always reflect the same data and can update it consistently. "Lifting state up" is a common pattern in React that helps manage state more efficiently across related components.

Why Do We Need to Lift State Up?

When two or more child components need to share the same state, keeping it inside one child causes issues. By lifting state up to a common parent, both child components can access and update the state easily.



What is "Lifting State Up" in React?

Example Without Lifting State Up (State in a Child Component)

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count
+ 1)}>Increase</button>
    </div>
  );
}

export default function App() {
  return (
    <div>
      <h1>Counter App</h1>
      <Counter />
      <Counter />
    </div>
  );
}
```

What's Wrong Here?

- Each Counter component has its own state.
- The counters don't sync—clicking on one button does not affect the other counter.
- We need to lift the state up so both components can share the same count.

What is "Lifting State Up" in React?

Solution: Lifting State Up

```
function Counter({ count, onIncrease }) {
  return (
    <div>
      <h2>Count: {count}</h2>
      <button
        onClick={onIncrease}>Increase</button>
    </div>
  );
}

export default function App() {
  const [count, setCount] = useState(0);

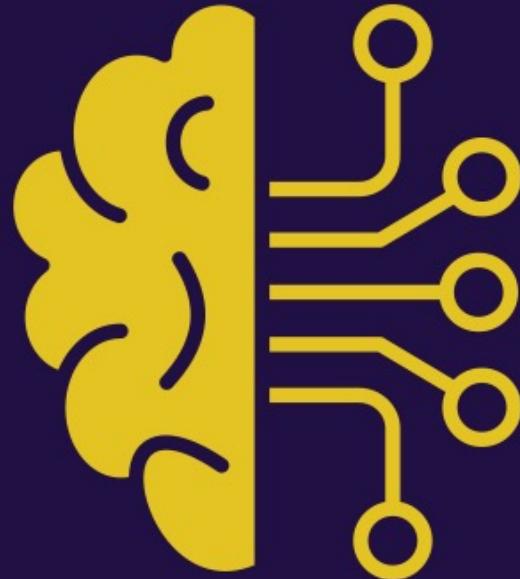
  return (
    <div>
      <h1>Counter App</h1>
      <Counter count={count} onIncrease={() =>
        setCount(count + 1)} />
      <Counter count={count} onIncrease={() =>
        setCount(count + 1)} />
    </div>
  );
}
```

Why This Works:

- The state is now in the parent (App), and both Counter components get the same value.
- Clicking any button updates the state in App, which re-renders both components.
- They stay in sync! 🎉

If the application becomes too complex, managing props using lifting state up becomes cumbersome.

How to share state across the App ?



We can share the state across the App primarily using two approaches:

- 1. Use React Context API (Best for Small to Medium Apps)**

The Context API allows data, functions to be shared across components without passing props manually.

- 2. Use State Management Libraries (Redux, Zustand)**

For larger applications, using a state management library like Redux helps manage state globally.

With these approaches, no need to pass props through each component. The required properties or data can be maintained at a centralized place and can be accessed in any component.

What is React Context API ?

React Context API is a way to share data between components without passing props manually at every level. It helps avoid props drilling, lifting state up making your code cleaner and easier to manage.

How to Use Context API ?

React Context API has 3 main steps:

1. Create a Context
2. Provide the Context (Wrap the Parent Component)
3. Consume the Context (Use the Data in Child Components)

Step 1: Create a context

We create a Context using `createContext()`. This creates a UserContext that will hold the user data.

```
import { createContext } from "react";
// Creating a Context
const UserContext = createContext();
```



What is React Context API ?

Step 2: Provide the Context (Wrap the Parent Component)

We wrap the components with `UserContext.Provider` and pass a value (`data`). Now, any child component inside `UserContext.Provider` can access the user data.

```
const App = () => {
  const user = { name: "Madan", age: 25 };

  return (
    <UserContext.Provider value={user}>
      <Parent />
    </UserContext.Provider>
  );
};

const Parent = () => <Child />;

const Child = () => <GrandChild />

export default App;
```



What is React Context API ?

Step 3: Consume the Context (Use the Data in Child Components)

We use `useContext()` hook to access the data inside components. Now, GrandChild can access user without props drilling!

```
import { useContext } from "react";
import { UserContext } from "./App"; // Import the context

const GrandChild = () => {
  const user = useContext(UserContext); // Access user data

  return <h1>User: {user.name}</h1>;
};

export default GrandChild;
```



useContext is a React Hook that lets you read and subscribe to context from your component. Below is the syntax,

```
const value = useContext(SomeContext);
```

- **SomeContext:** The context that you've previously created with `createContext`.

`useContext` returns the context value for the calling component. It is determined as the value passed to the closest `SomeContext.Provider` above the calling component in the tree.

If there is no such provider, then the returned value will be the `defaultValue` you have passed to `createContext` for that context. The returned value is always up-to-date.

React automatically re-renders components that read some context if it changes.



What is React Context API ?

Advanced: Updating Context with useState(): You can also update context values using useState().

```
const UserContext = createContext();

const App = () => {
  const [user, setUser] = useState({ name: "Madan", age: 30 });

  return (
    <UserContext.Provider value={{ user, setUser }}>
      <Child />
    </UserContext.Provider>
  );
};

const Child = () => <GrandChild />

const GrandChild = () => {
  const { user, setUser } = useContext(UserContext);
  return (
    <div>
      <h1>User: {user.name}</h1>
      <button onClick={() => setUser({ ...user, name: "John Doe" })}>
        Change Name
      </button>
    </div>
  );
};
```

What's Happening?

1. useState() manages the user state in App.
2. UserContext.Provider provides both user and setUser.
3. GrandChild uses useContext() to access and update the user data.

🚀 Now, clicking the button updates the user's name dynamically!

useReducer Hook

What is useReducer ?

- useReducer is a React Hook that helps manage complex state logic.
- It works like useState, but instead of directly updating the state, you use a reducer function.
- It is best suited when state updates involve multiple steps or depend on previous state.

Basic Syntax of useReducer

```
const [state, dispatch] = useReducer(reducer, initialArg, init?)
```

reducer: The reducer function that specifies how the state gets updated. It must be pure, should take the state and action as arguments, and should return the next state.

initialArg: The value from which the initial state is calculated.

optional init: The initializer function that should return the initial state. If it's not specified, the initial state is set to initialArg. Otherwise, the initial state is set to the result of calling init(initialArg).

useReducer returns an array with exactly two values:

The current state. During the first render, it's set to init(initialArg) or initialArg (if there's no init).
The dispatch function that lets you update the state to a different value and trigger a re-render.



useReducer Hook example

```
import { useReducer } from "react";

// Step 1: Define the reducer function
const counterReducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    default:
      return state;
  }
};

export default function Counter() {
  // Step 2: Use useReducer hook
  const [state, dispatch] = useReducer(counterReducer, { count: 0 });

  return (
    <div>
      <p>Count: {state.count}</p>
      {/* Step 3: Dispatch actions */}
      <button onClick={() => dispatch({ type: "INCREMENT" })}>+</button>
      <button onClick={() => dispatch({ type: "DECREMENT" })}>-</button>
    </div>
  );
}
```

useReducer Hook

useState vs useReducer

Feature	useState	useReducer
Best for	Simple state (e.g., form inputs, toggles)	Complex state logic (e.g., managing a shopping cart)
Updates state directly?	Yes, using <code>setState(newValue)</code>	No, uses <code>dispatch(action)</code> to update state
Handles state transitions?	No, you manually update values	Yes, state updates follow a structured reducer function
Code Organization	Multiple state update functions	Centralized logic with reducer. Improves Readability and Debugging
Performance	Function re-creations cause extra renders	Optimized (<code>dispatch</code> avoids extra renders)
Scalability	Harder to add more state changes	Easier to extend (just add cases in reducer)





Spring Security is a powerful framework that helps you protect your Spring applications.

- Handles authentication (who are you?) and authorization (what can you access?)
- Works seamlessly with Spring Boot
- Can protect both web and REST APIs
- Highly customizable



Why Use Spring Security?

- Protects Against common attacks like CSRF, CORs, Session Fixation
- Easy Login Systems: Adds username/password checks quickly.
- Customizable: You can tweak it to fit your needs
- Easily integrate with Databases, LDAP, OAuth2 / JWT / SAML



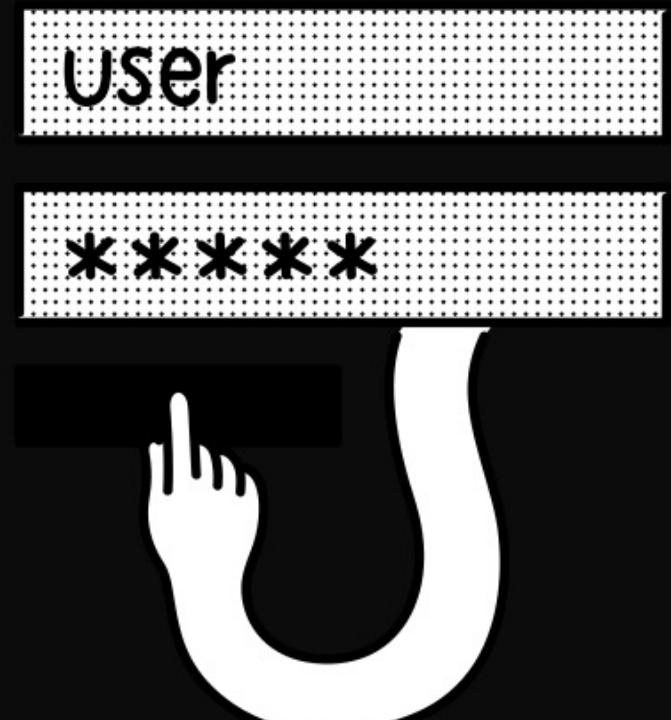
Default Behavior (Zero Configuration) of Spring Security

When you add this to your pom.xml:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

What happens automatically?

- A login form is shown
- A default user is created
- Console logs show the default password
- All URLs are secured by default



Default Spring Security configurations

By default, Spring Security framework protects all the paths present inside the web application. This behaviour is due to the code present inside the method `defaultSecurityFilterChain(HttpSecurity http)` of class `SpringBootWebSecurityConfiguration`

```
@Bean
@Order(SecurityProperties.BASIC_AUTH_ORDER)
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests((requests) -> requests.anyRequest().authenticated());
    http.formLogin(withDefaults());
    http.httpBasic(withDefaults());
    return http.build();
}
```

permitAll() Spring Security configurations

NOT RECOMMENDED
FOR PRODUCTION

We can permit all the requests coming towards our web application APIs, Paths using Spring Security framework like shown below,

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

        http.authorizeHttpRequests((requests) -> requests.anyRequest().permitAll())
            .formLogin(Customizer.withDefaults())
            .httpBasic(Customizer.withDefaults());
        return http.build();
    }
}
```

denyAll() Spring Security configurations

NOT RECOMMENDED
FOR PRODUCTION

We can deny all the requests coming towards our web application APIs, Paths using Spring Security framework like shown below,

```
@Configuration
public class ProjectSecurityConfig {

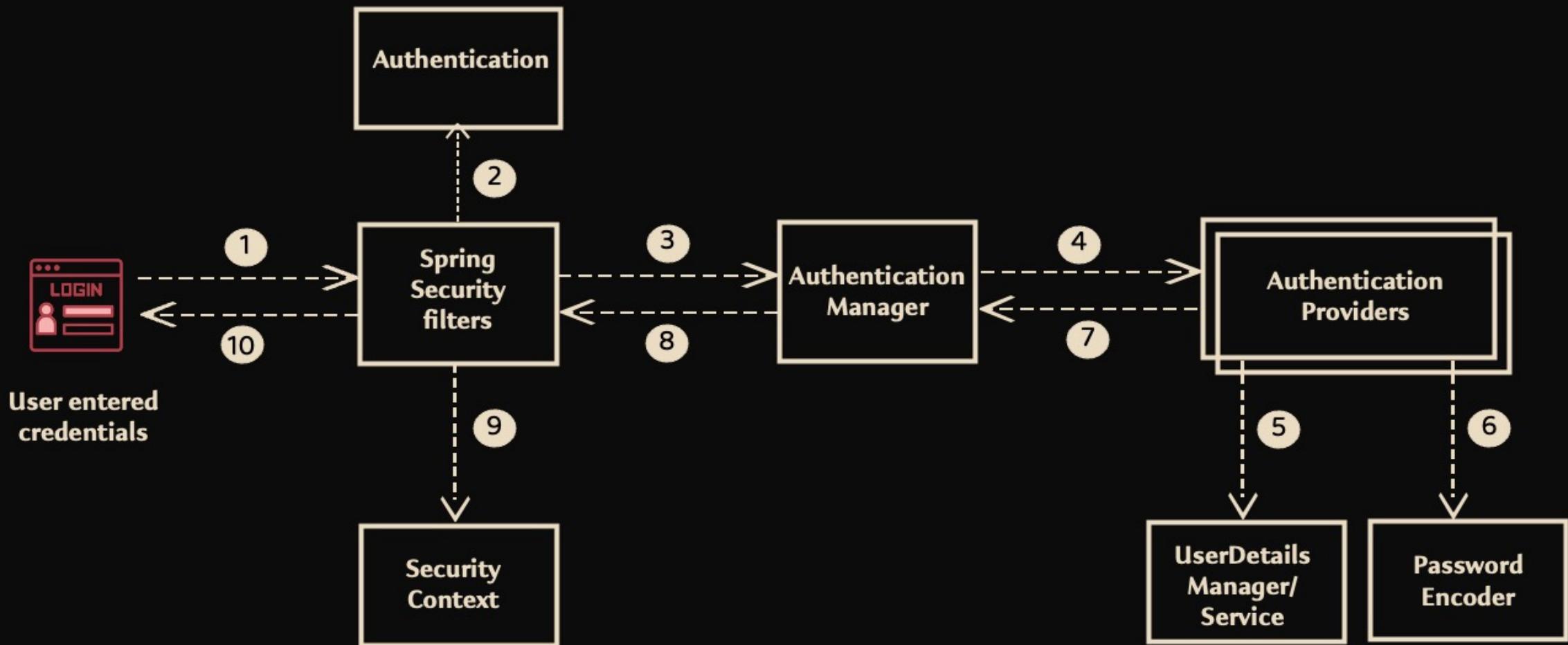
    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests((requests) -> requests.anyRequest().denyAll())
            .formLogin(Customizer.withDefaults())
            .httpBasic(Customizer.withDefaults());
        return http.build();
    }
}
```

Custom Spring Security configurations

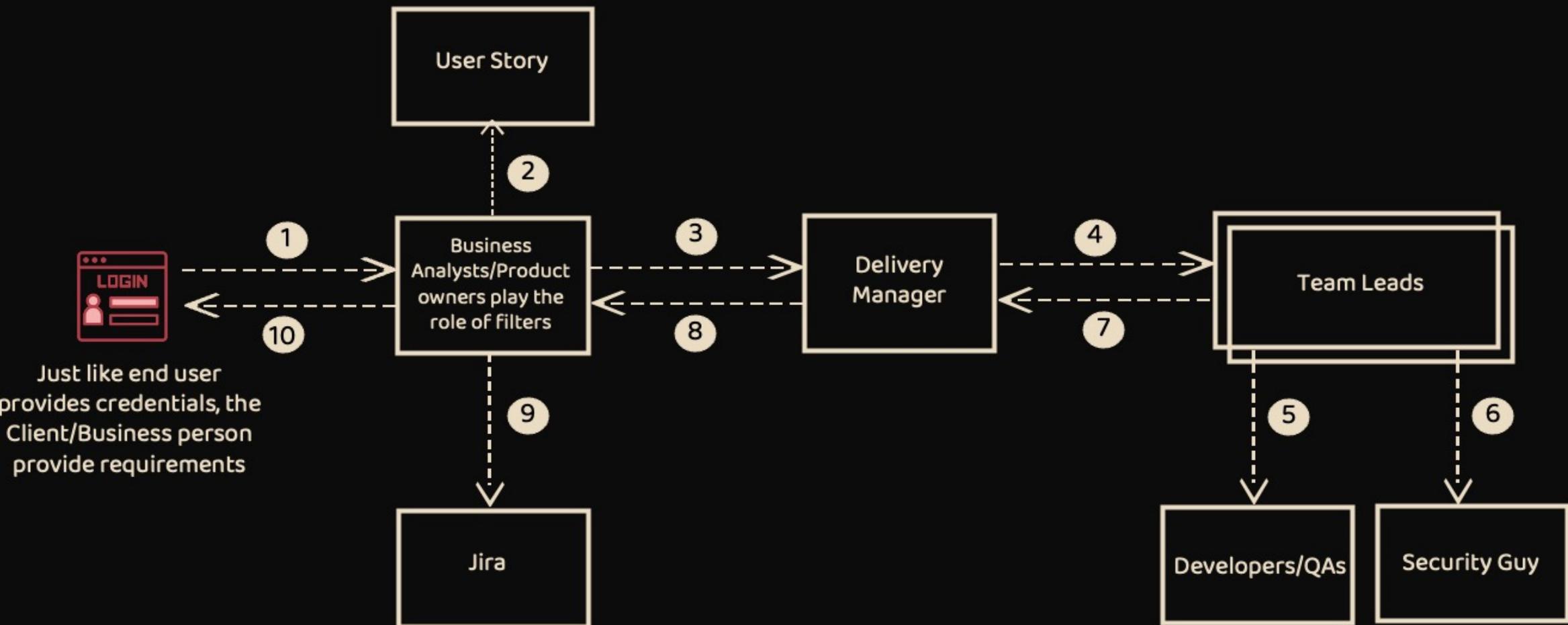
We can secure the web application APIs, Paths as per our custom requirements using Spring Security framework like shown below,

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http)
    throws Exception {
    return http.authorizeHttpRequests(requests) -> {
        publicPaths.forEach(path ->
            requests.requestMatchers(path).permitAll());
        requests.anyRequest().authenticated();
    }
    .formLogin(withDefaults())
    .httpBasic(withDefaults()).build();
}
```

Spring Security Internal flow



Analogy of Spring Security Internal flow with SDLC



★ Spring Security Filters

A series of Spring Security filters intercept each request & work together to identify if Authentication is required or not. If authentication is required, accordingly navigate the user to login page or use the existing details stored during initial authentication.

★ Authentication

Filters like `UsernamePasswordAuthenticationFilter` will extract username/password from HTTP request & prepare Authentication type object. Because Authentication is the core standard of storing authenticated user details inside Spring Security framework.

★ AuthenticationManager

Once received request from filter, it delegates the validating of the user details to the authentication providers available. Since there can be multiple providers inside an app, it is the responsibility of the AuthenticationManager to manage all the authentication providers available. In simple words, the authentication manager takes the responsibility for authentication.

★ AuthenticationProvider

AuthenticationProviders has all the core logic of validating user details for authentication.

★ UserDetailsService/UserDetailsManager

UserDetailsService/UserDetailsManager helps in retrieving, creating, updating, deleting the User Details from the DB/storage systems.

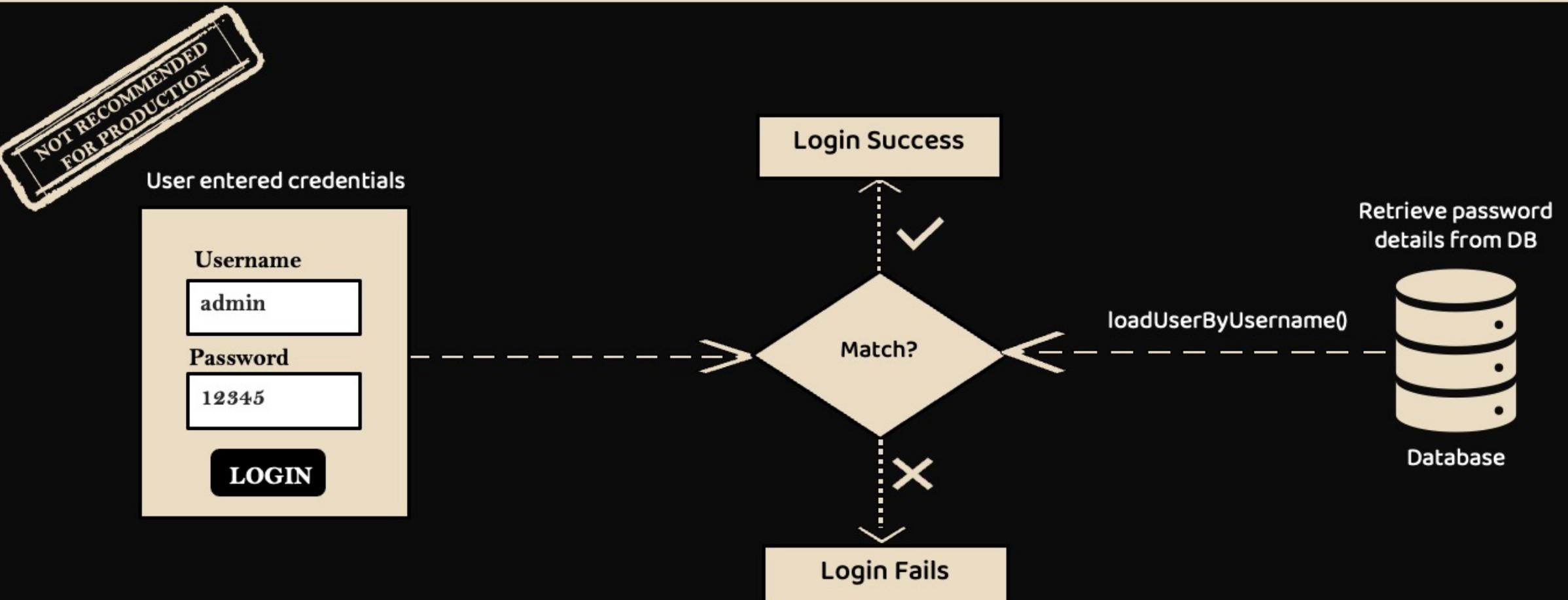
★ PasswordEncoder

Service interface that helps in encoding & hashing passwords. Otherwise we may have to live with plain text passwords ☹

★ SecurityContext

Once the request has been authenticated, the Authentication will usually be stored in a thread-local SecurityContext managed by the `SecurityContextHolder`. This helps during the upcoming requests from the same user.

How passwords validated with plain passwords



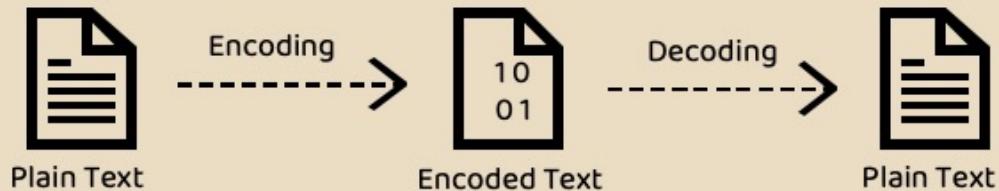
Storing the passwords in a plain text inside a storage system like DB will have Integrity & Confidentiality issues. So this is not a recommended approach for Production applications.

Different ways for Data privacy

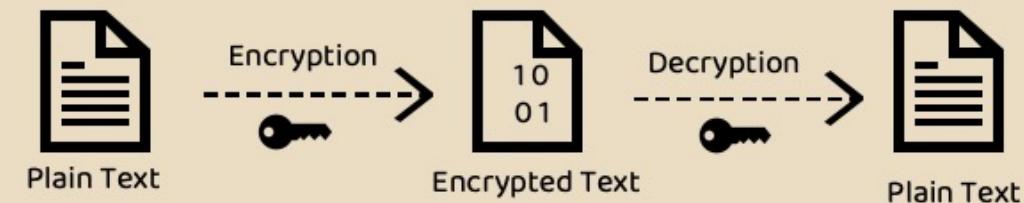
Encoding	Encryption	Hashing
✓ Encoding is defined as the process of converting data from one form to another and has nothing to do with cryptography.	✓ Encryption is defined as the process of transforming data in such a way that guarantees confidentiality.	✓ In hashing, data is converted to the hash value using some hashing function.
✓ It involves no secret and completely reversible.	✓ To achieve confidentiality, encryption requires the use of a secret which, in cryptographic terms, we call a "key".	✓ Data once hashed is non-reversible. One cannot determine the original data from a hash value generated.
✓ Encoding can't be used for securing data. Below are the various publicly available algorithms used for encoding. Ex: ASCII, BASE64, UNICODE	✓ Encryption can be reversible by using decryption with the help of the "key". As long as the "key" is confidential, encryption can be considered as secured.	✓ Given some arbitrary data along with the output of a hashing algorithm, one can verify whether this data matches the original input data without needing to see the original data

Encoding Vs Encryption Vs Hashing

Encoding & Decoding



Encryption & Decryption

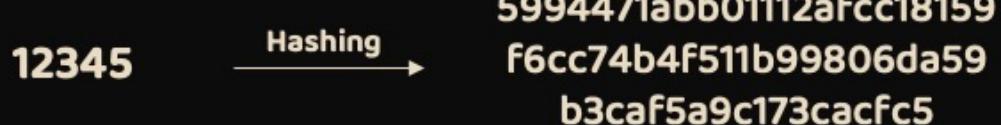


Hashing (Only 1-way)



01

Hash functions take any data as input and produce a unique string of bytes in return. Given the same input, the hash function always reproduces the same string of bytes. The output of a hash function is often called a digest or a HASH.

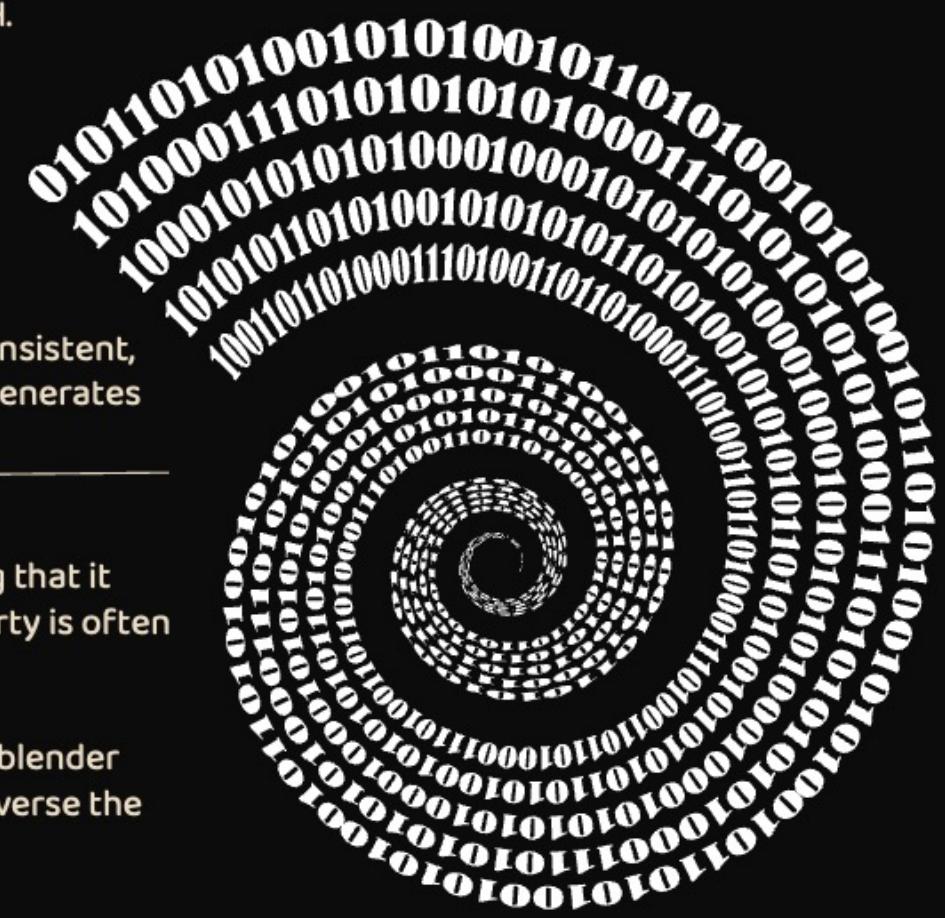


A hash function transforms input of any length (such as a file, message, or video) into a consistent, fixed-length output (e.g., 256 bits for SHA-256). When the same input is hashed, it always generates the same resulting digest or hash.

02

A fundamental characteristic of a hash function is its irreversibility, meaning that it should not be possible to deduce the input from its output alone. This property is often referred to as being "one-way."

In other words, given the output produced by a hash function (depicted as a blender here), it should be computationally infeasible, or practically impossible, to reverse the process and uncover the original input.



Hashing passwords

The typical approach for storing passwords within an organization involves hashing user passwords and retaining only the resulting digests. When a user attempts to log in to your website, the process typically follows these steps:

1. You receive the user's password during the registration process.
2. You hash the provided password and discard the original. Store it in a storage system like Database
3. During login operation, you compare the resulting digest with the stored version; if they match, the user gains access.

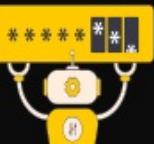
While this may look like a perfect solution as no one can get to know the plain password from the hash or digest value even if they manage to steal the user data from DB. It has some loop holes. Let's imagine few users they used simple passwords and there is also a good chance that multiple user choosed to have the same password like 123456 or password etc. Below is how, it is going to look,

User ID	Username	Plain password	Hash password
1	johndoe	12345	5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5
2	mike123	password	5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
3	pavan189	12345	5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5
4	saanvi180	password	5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8

There will be couple of problems if we use simple hashing mechanisms for passwords management.

1. If an attacker gains access to hashed passwords in a DB, they could potentially launch a brute force attack or an exhaustive search (on Rainbow table) by trying all possible passwords. This method tests each attempt against the entire database of hashed passwords. Ideally, we aim to limit attackers to targeting only one hashed password at a time.
2. Hash functions are designed to be fast, which attackers can exploit to try numerous passwords per second during a brute force attack. Ideally, we should implement mechanisms to impede such attacks, slowing down the process significantly.

What is a Brute force attack ?



A brute force attack is a trial-and-error method used by hackers to decode encrypted data, such as passwords or encryption keys. In this type of attack, the attacker systematically tries all possible combinations of characters until the correct one is found. It's essentially like trying every possible key in a lock until the right one opens it.

What is a Rainbow table or Dictionary table attack ?



A rainbow table or Dictionary table attacks are a type of cryptographic attack used to crack password hashes. It involves precomputing and storing a large number of password hashes along with their corresponding plaintext passwords in a table known as a "rainbow table" or a "dictionary table".

When a hashed password is obtained, instead of computing hashes for all possible passwords as in a brute force attack, the attacker can simply look up the hash in the rainbow or dictionary table to find the corresponding plaintext password. This significantly reduces the time and computational resources required to crack the password.

Hashing passwords

Let's try to understand how to overcome the challenges of Brute force attacks and Rainbow table attacks,

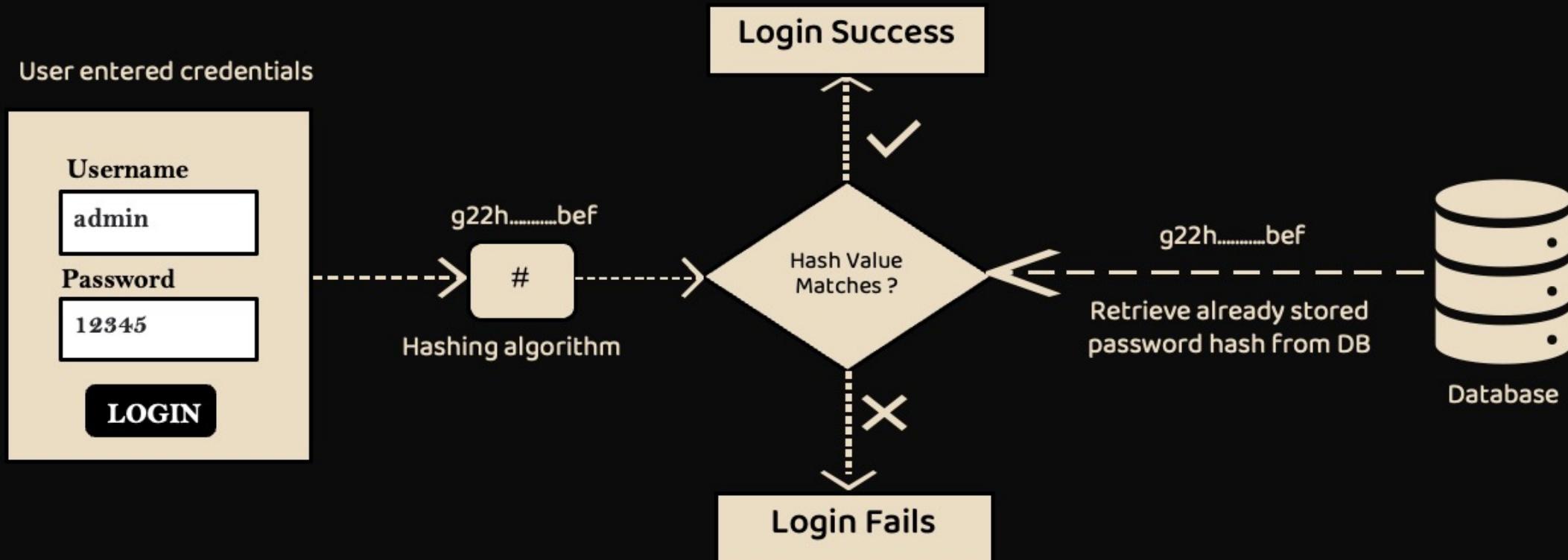
To prevent rainbow table attacks, **Salts** are commonly used. Salts are random values that are unique for each user and are made public. The salt value is stored as part of the resulting hash itself. Specifically, the salt is included in the hash string along with the actual hashed password.

The salt is typically stored at the beginning of the hash string and is used during the password verification process to generate the same hash again for comparison. Since each user has a unique hash function, attackers can't precompute large tables of passwords (rainbow tables) to test against the entire database of stolen password hashes.

Plain text password + Random salt value + Hashing Algo = Strong Hash Value

Spring Security provides industry recommended Password Encoders that are capable of generating random salt and leverage password hashing algorithms like bcrypt

How passwords validated with hashed passwords



Storing & managing the passwords with hashing is the recommended approach for Production applications. With various PasswordEncoders available inside Spring Security, it makes our life easy.

Methods inside PasswordEncoder Interface

```
public interface PasswordEncoder {  
  
    String encode(CharSequence rawPassword);  
  
    boolean matches(CharSequence rawPassword, String encodedPassword);  
  
    default boolean upgradeEncoding(String encodedPassword) {  
        return false;  
    }  
}
```

Different implementations of PasswordEncoder inside Spring Security

★ **NoPasswordEncoder** (Not recommended for Prod apps)

★ **StandardPasswordEncoder** (Not recommended for Prod apps)

★ **Pbkdf2PasswordEncoder**

★ **BCryptPasswordEncoder**

★ **SCryptPasswordEncoder**

★ **Argon2PasswordEncoder**

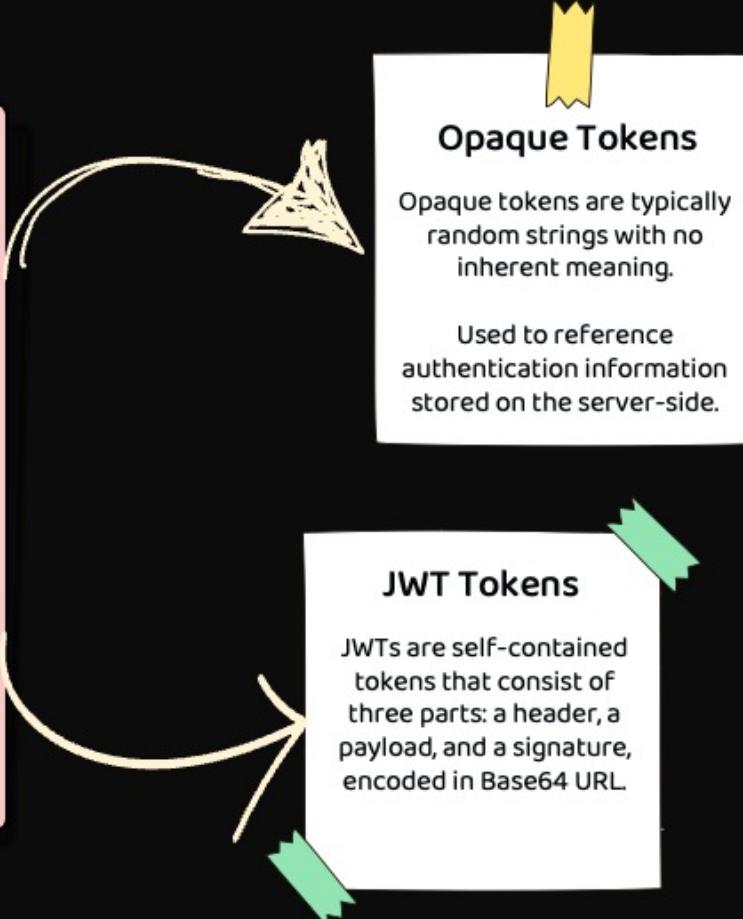
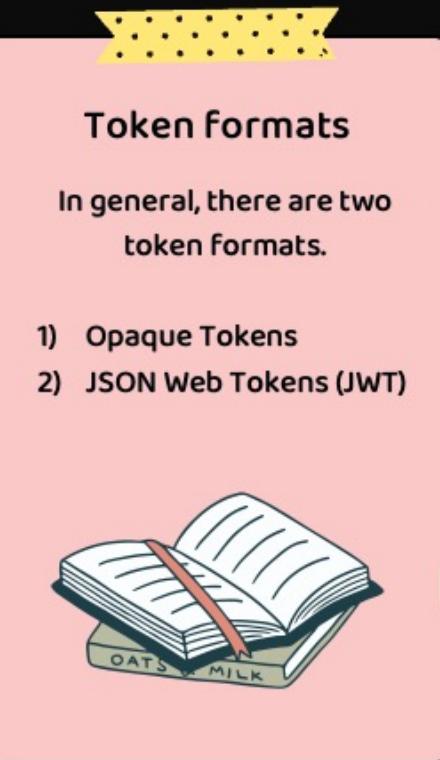
Configure users using InMemoryUserDetailsManager

NOT RECOMMENDED
FOR PRODUCTION

Instead of defining a single user inside application.properties, we can define multiple users along with their authorities with the help of **InMemoryUserDetailsManager & UserDetails**

```
@Bean
public UserDetailsService userDetailsService() {
    var user1 = User.builder().username("madan")
        .password("$2a$12$vsi964EvG80PyZfayERimu1jX7rbTNiji7SaGr8waXq3zwZX2kXqm")
        .roles("USER").build();
    var user2 = User.builder().username("admin")
        .password("$2a$12$neiGI7viF8rIbgnWruvNZSQNqtG4F3Mt3631ehox0BEk/N9v0")
        .roles("USER", "ADMIN").build();
    return new InMemoryUserDetailsManager(user1, user2);
}

@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```



- ✓ JWT means JSON Web Token. It is a token implementation which will be in the JSON format and designed to use for the web requests.
- ✓ JWT is the most common and favorite token type that many systems use these days due to its special features and advantages.
- ✓ JWT tokens can be used both in the scenarios of Authorization/Authentication along with Information exchange which means you can share certain user related data in the token itself which will reduce the burden of maintaining such details in the sessions on the client/server side.

A JWT token has 3 parts each separated by a period(.) Below is a sample JWT token,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWliOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvAG4gRG9IiwiWF0IjoxNTE2MjM5MDIyFQ.SFIKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

1. Header
2. Payload
3. Signature (Optional)

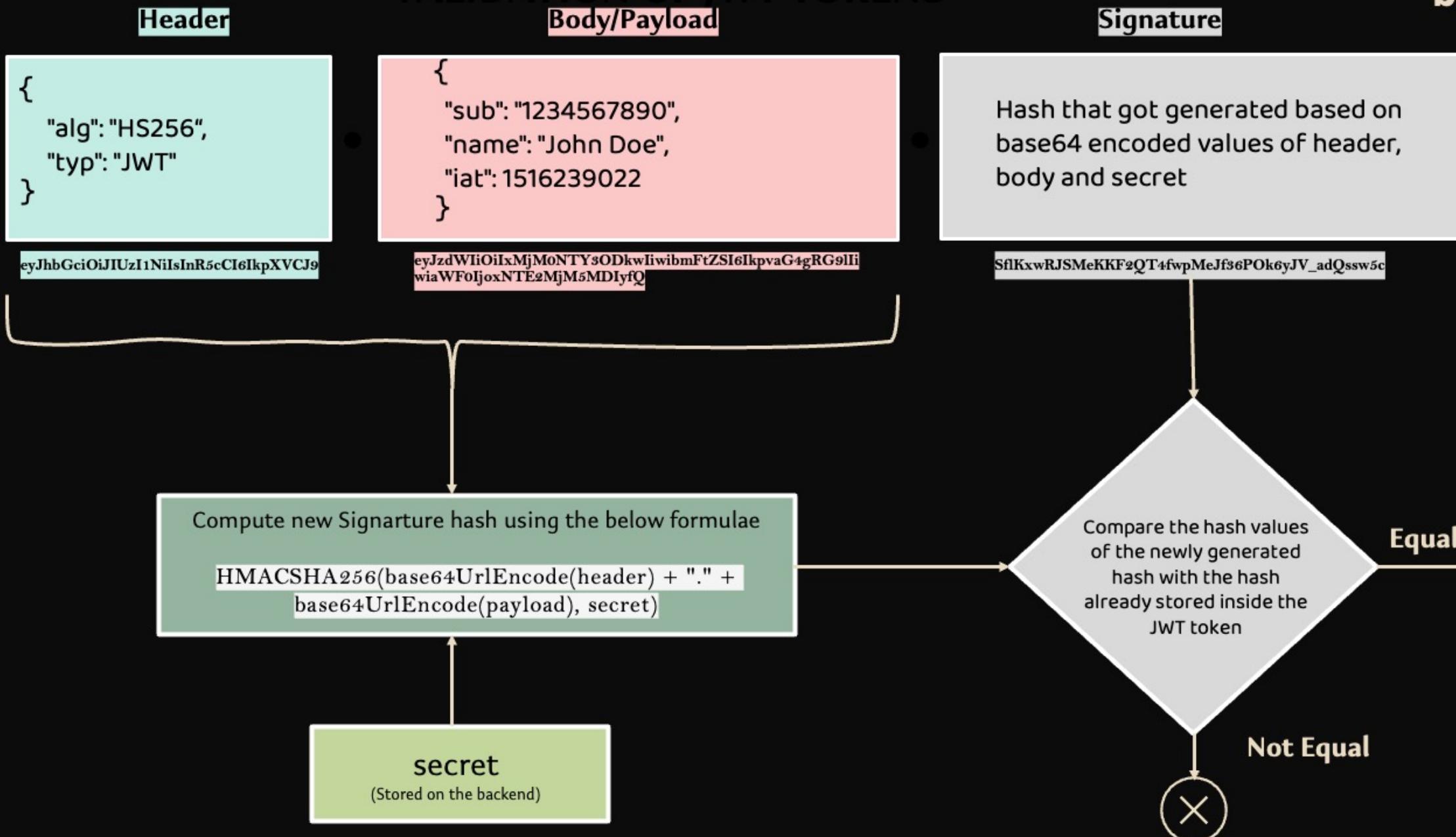
- ✓ The last part of the token is the digital signature. This part can be optional if the party that you share the JWT token is internal and that someone who you can trust but not open in the web.
- ✓ But if you are sharing this token to the client applications which will be used by all the users in the open web then we need to make sure that no one changed the header and body values like Authorities, username etc.
- ✓ To make sure that no one tampered the data on the network, we can send the signature of the content when initially the token is generated. To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

- ✓ For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

- ✓ The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

VALIDATION OF JWT TOKENS



Client-Side Route Protection in React

What we want:

- Secure certain routes (like /checkout)
- Only allow access if user is authenticated
- Redirect unauthenticated users to /login

What is Client-Side Route Protection?

Client-side route protection means:

- Checking user's auth state before rendering a route
- Preventing users from accessing protected components unless they're logged in
- Usually done using React Router + Context API

How we are going to secure routes on React app ?

- auth-context.jsx: Handles auth state globally (like JWT and user)
- ProtectedRoute.jsx: Wrapper that checks authentication before rendering
- App.jsx + Router: Contains both public and protected routes



A context focusing Auth will be created,

```
export const AuthContext = createContext();
export const useAuth = () => useContext(AuthContext);
```

AuthContext provides following:

```
{
  isAuthenticated,
  user,
  jwtToken,
  loginSuccess(jwtToken, user),
  logout()
}
```

How it's used:

- **AuthProvider** wraps the app
- Any component can use **useAuth()** to get login/logout state

Persisting Auth with LocalStorage

On login:

- Save JWT and user in localStorage
- Update context state

On logout:

- Clear localStorage
- Update context state

```
useEffect(() => {
  if (authState.isAuthenticated) {
    localStorage.setItem("jwtToken", authState.jwtToken);
    localStorage.setItem("user", JSON.stringify(authState.user));
  } else {
    localStorage.removeItem("jwtToken");
    localStorage.removeItem("user");
  }
}, [authState]);
```

Creating a Protected Route

ProtectedRoute.jsx (simplified):

```
import { Navigate, Outlet } from "react-router-dom";
import { useAuth } from "../store/auth-context";

const ProtectedRoute = () => {
  const { isAuthenticated } = useAuth();
  return isAuthenticated ? <Outlet /> : <Navigate to="/login" />;
};

export default ProtectedRoute;
```

Wrap protected paths inside <ProtectedRoute>
Uses <Outlet /> to render nested route content

Defining Routes with Protection

```
<Route path="/" element={<App />} errorElement={<ErrorPage />}>
  <Route index element={<Home />} loader={productsLoader} />
  <Route path="/login" element={<Login />} action={loginAction} />
  ...
  <Route element={<ProtectedRoute />}>
    <Route path="/checkout" element={<CheckoutForm />} />
  </Route>
</Route>
```

/checkout will only be rendered if authenticated
Else, it redirects to /login

What are Derived Query Methods ?

Derived query methods are auto-generated queries based on method names in Spring Data JPA.

Instead of writing SQL queries manually, you just name a method in your repository, and Spring Data JPA will automatically figure out the query for you.

How Do Derived Query Methods Work?

Spring Data JPA understands method names and automatically generates queries based on specific keywords in the method name.

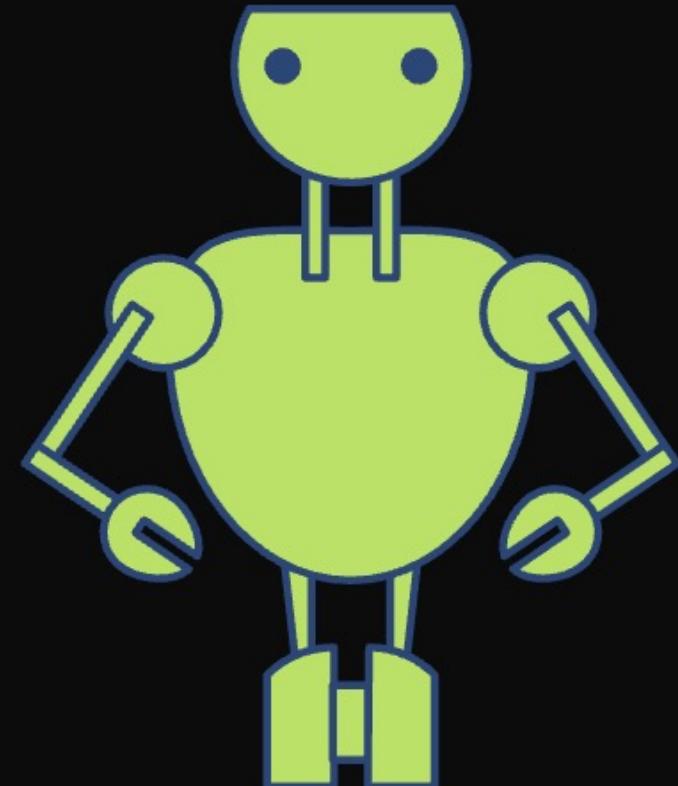
For example, with the below method defined,

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    // Spring will automatically generate the SQL query for this method!
    Optional<Customer> findByEmail(String email);
}
```

This will automatically generate and execute:

```
SELECT * FROM customers WHERE email = ?;
```



Do you know ?

A derived query method in Spring Data JPA consists of two main components separated by the first **By** keyword:

Introducer Clause

- This part specifies the type of operation to be performed on the database.
- Common introducers include `find`, `read`, `query`, `count`, and `get`.
- You can also include `Distinct` to ensure unique results in the query.

Criteria Clause

- This part begins after the first `By` keyword and defines conditions on entity properties.
- Conditions can be combined using `And` and `Or` keywords.

Example Usage:

- `findByEmail(String email)`
- `readByEmail(String email)`
- `queryByEmail(String email)`
- `countByEmail(String email)`

Using `readBy`, `getBy`, and `queryBy` instead of `findBy` does not change the behavior of the query.



Derived Query Methods in Spring Data JPA

Common Keywords in Derived Query Methods

Keyword	Meaning	Example method name	Generated SQL
findBy	Find records by a field	findByDepartment(String dept)	SELECT * FROM customers WHERE department = ?
countBy	Count records by a field	countByDepartment(String dept)	SELECT COUNT(*) FROM customers WHERE department = ?
existsBy	Check if record exists	existsByName(String name)	SELECT CASE WHEN COUNT(*) > 0 THEN TRUE ELSE FALSE END FROM customers WHERE name = ?
deleteBy	Delete records by a field	deleteByName(String name)	DELETE FROM customers WHERE name = ?
removeBy	Remove records by a field	removeByAge(int age)	DELETE FROM customers WHERE age = ?

Using Multiple Fields in Queries

Example method name	Generated SQL
findByNameAndAge(String name, int age)	SELECT * FROM customers WHERE name = ? AND age = ?
findByDepartmentOrAge(String dept, int age)	SELECT * FROM customers WHERE department = ? OR age = ?
findDistinctByNameAndAge(String name, int age)	SELECT DISTINCT(*) FROM customers WHERE name = ? AND age = ?

Derived Query Methods in Spring Data JPA

Using Comparisons in Queries

Example method name	Generated SQL
findByAgeGreaterThan(int age)	SELECT * FROM customers WHERE age > ?;
findByAgeLessThan(int age)	SELECT * FROM customers WHERE age < ?;
findByAgeBetween(int start, int end)	SELECT * FROM customers WHERE age BETWEEN ? AND ?;
findByNameLike(String pattern)	SELECT * FROM customers WHERE name LIKE ?;
findByNameNotLike(String pattern)	SELECT * FROM customers WHERE name NOT LIKE ?;
findByNameStartingWith(String prefix)	SELECT * FROM customers WHERE name LIKE ?%;
findByNameEndingWith(String suffix)	SELECT * FROM customers WHERE name LIKE %?;
findByNameContaining(String substring)	SELECT * FROM customers WHERE name LIKE %?%;

Sorting with Derived Queries

Example method name	Generated SQL
findByDepartmentOrderByAgeAsc(String dept)	SELECT * FROM customers WHERE department = ? ORDER BY age ASC
findByDepartmentOrderByNameDesc(String dept)	SELECT * FROM customers WHERE department = ? ORDER BY name DESC

Derived Query Methods in Spring Data JPA

Limiting the Number of Results

Example method name	Generated SQL
findFirstByOrderByAgeAsc()	SELECT * FROM customers ORDER BY age ASC LIMIT 1
findTop3ByOrderByAgeDesc()	SELECT * FROM customers ORDER BY age DESC LIMIT 3

Handling Null Values

Example method name	Generated SQL
findByDepartmentIsNull()	SELECT * FROM customers WHERE department IS NULL
findByDepartmentIsNotNull()	SELECT * FROM customers WHERE department IS NOT NULL

Working with Dates

Example method name	Generated SQL
findByStartDateAfter(Date startDate)	SELECT * FROM customers WHERE start_date > ?
findByStartDateBefore(Date startDate)	SELECT * FROM customers WHERE start_date < ?

Derived Query Methods in Spring Data JPA

Checking multiple values

Example method name	Generated SQL
findByDepartmentIn(Arrays.asList("HR", "IT"))	SELECT * FROM customers WHERE department IN ('HR', 'IT')
findByDepartmentNotIn(Arrays.asList("HR", "IT"))	SELECT * FROM customers WHERE department NOT IN ('HR', 'IT')

Checking Boolean values

Example method name	Generated SQL
findByActiveTrue()	SELECT * FROM customers WHERE active = true
findByActiveFalse()	SELECT * FROM customers WHERE active = false

Ignoring case

Example method name	Generated SQL
findByFirstnameIgnoreCase(String name)	SELECT * FROM customers U WHERE UPPER(U.name) = UPPER(?)

Combining Everything: A Complex Query

You can mix multiple conditions together!

Example:

```
List<Customer> findByDepartmentAndAgeGreaterThanOrEqualToAsc(String department, int age);
```

This will generate:

```
SELECT * FROM customers WHERE department = ? AND age > ? ORDER BY name ASC;
```

If a derived query gets too complex, it's
better to use **@Query**

CompromisedPasswordChecker

RECOMMENDED
FOR PRODUCTION

Using CompromisedPasswordChecker implementation, we can check if a password has been compromised. This feature is released as part of Spring Security 6.3 version.

```
@Bean
public CompromisedPasswordChecker compromisedPasswordChecker() {
    return new HaveIBeenPwnedRestApiPasswordChecker();
}
```

```
CompromisedPasswordDecision decision =
    compromisedPasswordChecker.check(registerRequestDto.getPassword());
if(decision.isCompromised()) {
    return ResponseEntity
        .status(HttpStatus.BAD_REQUEST)
        .body(Map.of("password", "Choose a strong password"));
}
```

Understanding Relationships in Spring Data JPA

In real-world apps, data is often connected (e.g., a customer has an address, a book has an author). JPA **relationships** help model these connections between entities (tables) in your database.

- JPA relationships define how one entity connects to another
- They help in maintaining data consistency and navigating between entities easily
- These relationships mimic foreign key relationships in databases
- Without relationships, Developers need to write complex join queries

Spring Data JPA supports four main types of relationships:

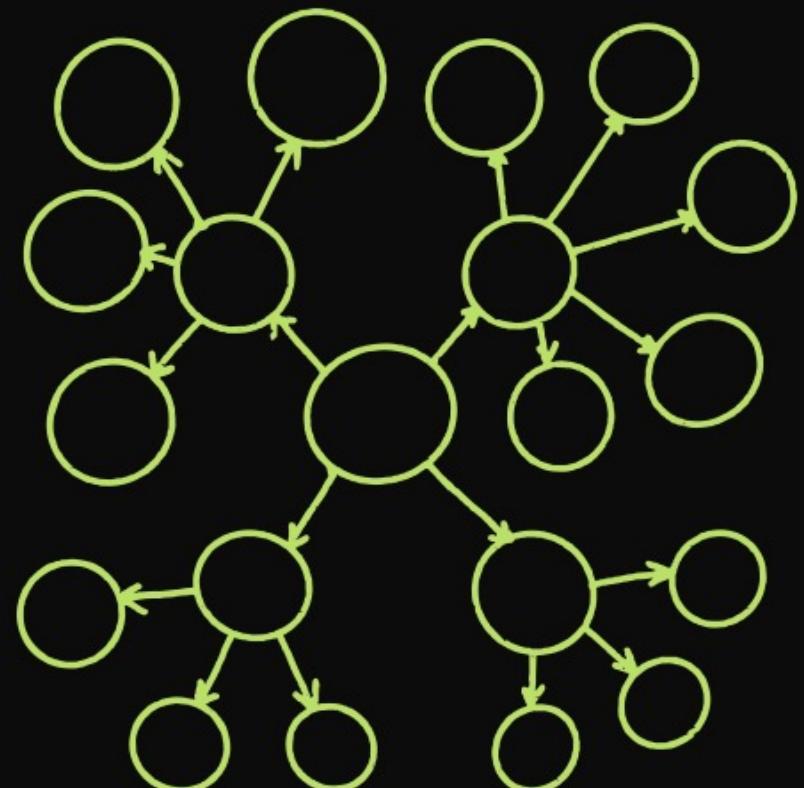
OneToOne: One entity is linked to exactly one other entity.

OneToMany: One entity is linked to multiple entities.

ManyToOne: Multiple entities are linked to one entity.

ManyToMany: Multiple entities are linked to multiple entities.

Each relationship has a specific use case and way to implement.

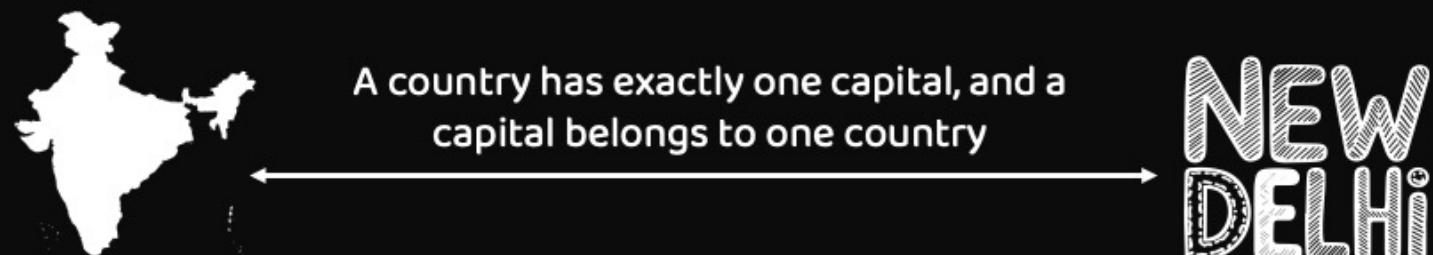
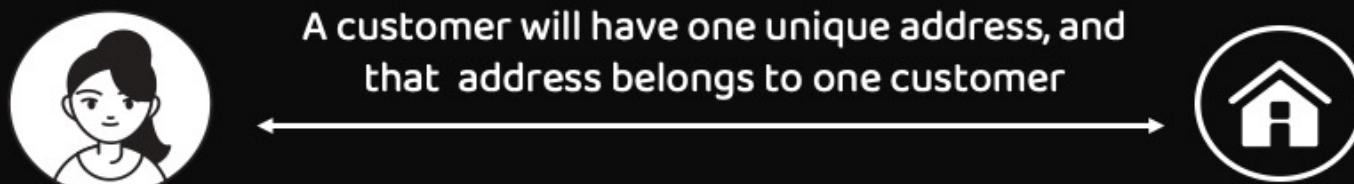
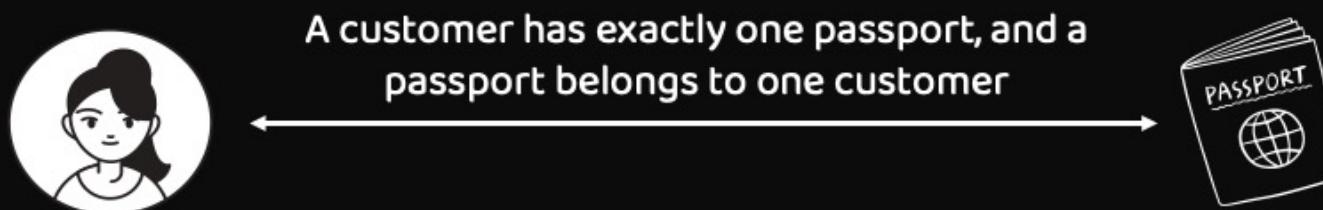


OneToOne Relationship in Spring Data JPA

What is a OneToOne Relationship?

A single record in one table is linked to a single record in another table. Used when two entities have a unique, exclusive connection

Real-World Examples:



OneToOne Relationship example

The Customer and Address entities are connected using a OneToOne relationship, with Address owning the relationship (holding the foreign key).

Customer.java

```
@OneToOne(mappedBy = "customer", cascade = CascadeType.ALL)
private Address address;
```

Address.java (Owning side)

```
@OneToOne(fetch = FetchType.EAGER, optional = false)
@JoinColumn(name = "customer_id", nullable = false)
@OnDelete(action = OnDeleteAction.CASCADE)
private Customer customer;
```

Key Annotations:

@OneToOne: Defines the OneToOne relationship between entities.

@OneToOne(mappedBy = "customer", cascade = CascadeType.ALL): Indicates Customer is the non-owning side. The customer field in the Address entity defines the relationship. Meaning: Customer doesn't store a foreign key; it relies on Address to manage the link. With cascade setting, we are telling When ever save/update/delete Customer happening, the same operation needs to cascaded to Address.

@OneToOne(fetch = FetchType.EAGER, optional = false):

fetch = FetchType.EAGER: Loads the Customer along with the Address record

optional = false: Ensures every Address must have a Customer (no nulls allowed).

@JoinColumn(name = "customer_id", nullable = false): Specifies the foreign key column in the owning table.

@OnDelete(action = OnDeleteAction.CASCADE): If a Customer is deleted, the associated Address is automatically deleted.

Why Is This a Bidirectional Relationship?

A bidirectional relationship means both entities can access each other.

- Customer can access its Address (via `customer.getAddress()`).
- Address can access its Customer (via `address.getCustomer()`).

Understanding FETCH Types in JPA

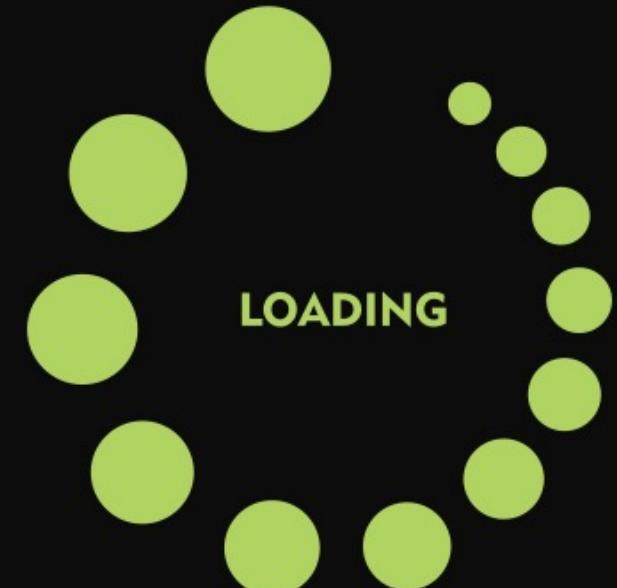
What Are Fetch Types in JPA?

Fetch Types define when related entities are loaded from the database. When you define relationships like `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`, JPA needs to know:

Should the related data be loaded immediately when the parent is loaded?

Or should it be loaded later, only when needed?

That's where `fetch = FetchType.LAZY` or `fetch = FetchType.EAGER` comes in.



Fetch Type	Meaning	Behavior
EAGER	Fetch immediately	Loads the related entity right away
LAZY	Fetch when accessed (on demand)	Loads the related entity only when used

Understanding FETCH Types in JPA

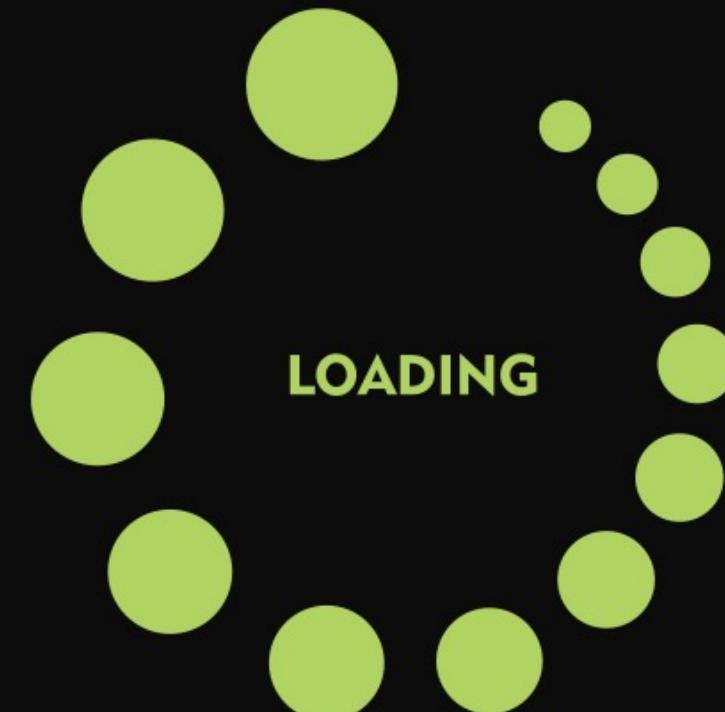
Think of It Like Ordering Food 🍕

EAGER = You order the main dish + side dish at the same time
→ Even if you're not hungry for the side, it still comes

LAZY = You order only the main dish
→ You ask for the side dish only if/when you want it

Default Fetch Types

Relationship Annotation	Default Fetch Type
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY



What Are Cascade Types in JPA?

Cascade in JPA means:

"If I do something to the parent entity, automatically do it to the child entity too."

Think of it like a chain reaction – changes made to one entity get automatically applied to related entities.

Real-World Analogy

Imagine a Customer and their Address:

If a customer is saved, their address should be saved too.

If the customer is deleted, their address should be deleted too.

That's cascading: you don't need to save or delete related objects separately.

Why Do We Use Cascade?

- To reduce boilerplate code
- To avoid manually saving/deleting related entities
- To keep entities in sync automatically



Understanding Cascade Types in JPA

Available Cascade Types

Cascade Type	What it does
PERSIST	Saves the child entity when the parent is saved
MERGE	Updates the child entity when the parent is updated
REMOVE	Deletes the child entity when the parent is deleted
REFRESH	Refreshes the child when parent is refreshed from DB
DETACH	Detaches the child from the persistence context when parent is detached
ALL	Applies all the above actions

Code Example – CascadeType.PERSIST

```
@OneToOne(cascade = CascadeType.PERSIST)  
private Address address;
```

```
Customer customer = new Customer();  
customer.setName("John");  
  
Address address = new Address();  
address.setCity("Delhi");  
address.setCustomer(customer);  
  
customer setAddress(address);  
customerRepository.save(customer); // Address is saved too!
```

What is @OnDelete(action = OnDeleteAction.CASCADE)?

@OnDelete is an annotation in Hibernate/JPA (Java).

Purpose: Tells the database to automatically delete related records when a parent record is deleted.

Uses ON DELETE CASCADE at the database level (e.g., in SQL).

Code Example

```
@OneToOne(fetch = FetchType.EAGER, optional = false)
@OnDelete(action = OnDeleteAction.CASCADE)
@JoinColumn(name = "customer_id", nullable = false)
private Customer customer;
```

If a Customer is deleted, the associated Address record is automatically deleted by the database.

The enum `org.hibernate.annotations.OnDeleteAction` defines the strategy Hibernate should apply when a parent entity is deleted, in relation to its child/associated entities – directly in the database (not through Java).

OnDeleteAction Values

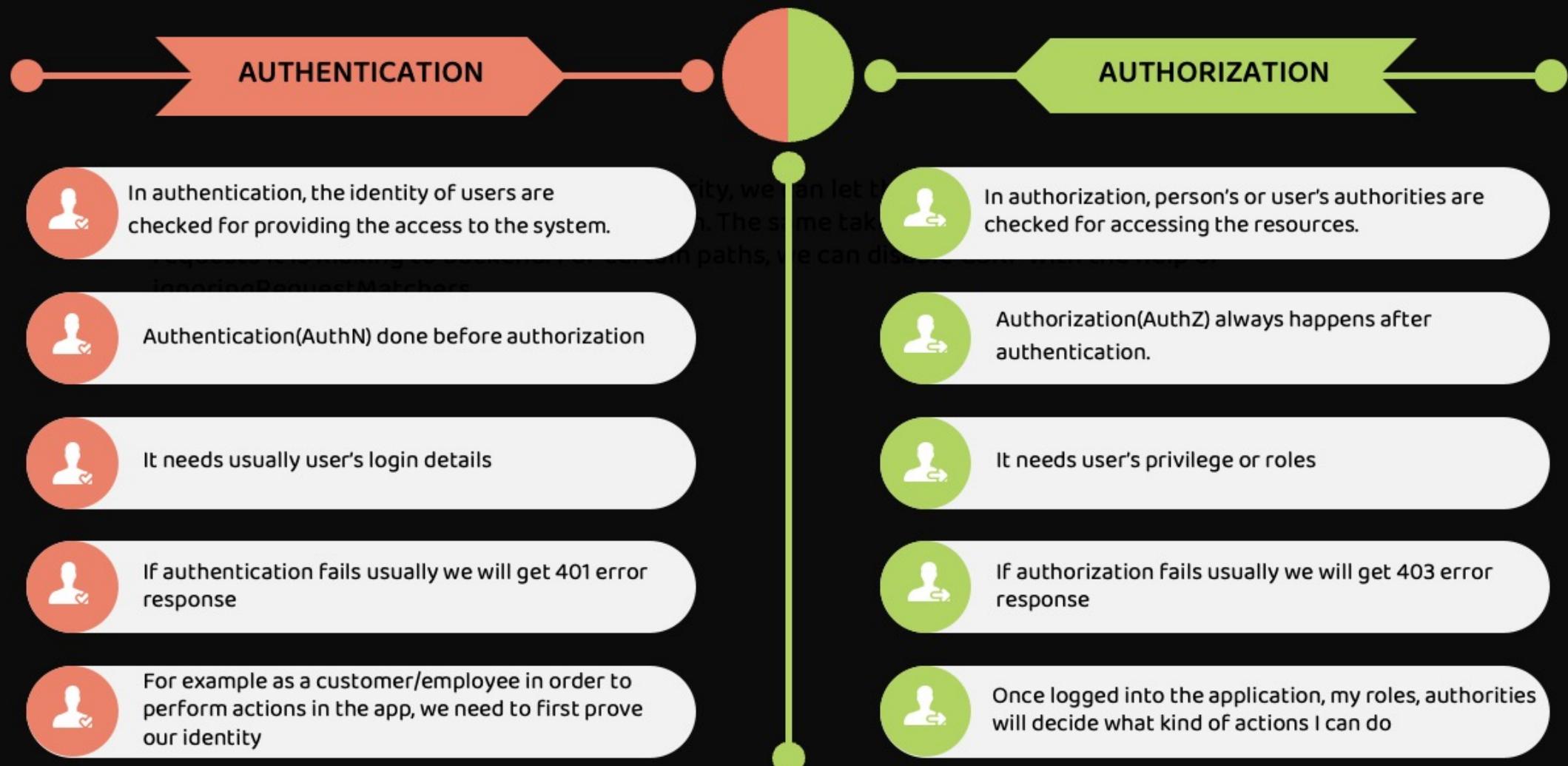
Enum Value	Description
<code>NO_ACTION</code>	Default behavior. If a parent is deleted while children still exist, the database raises a foreign key constraint violation (maybe later in transaction).
<code>CASCADE</code>	Automatically deletes the child rows when the parent row is deleted (via ON DELETE CASCADE at DB level).
<code>RESTRICT</code>	Immediately blocks the deletion of the parent if any child exists (stricter than <code>NO_ACTION</code>).
<code>SET_NULL</code>	Sets the child's foreign key to NULL when the parent is deleted (child remains, but unlinked).
<code>SET_DEFAULT</code>	Sets the child's foreign key to its default value (if any) when the parent is deleted.

@OnDelete vs. CascadeType.REMOVE

Feature	@OnDelete	CascadeType.REMOVE
Where it runs	Database level (SQL foreign key)	Application level (Java/Hibernate)
How it works	Database deletes child records automatically	Hibernate deletes child records in code
Performance	Faster (database handles it)	Slower (Hibernate queries each record)
Code example	<code>@OnDelete(action = OnDeleteAction.CASCADE)</code>	<code>cascade = CascadeType.REMOVE</code>
When to use	When you want database to manage deletions	When you need app-level control

Key Difference: `@OnDelete` is a database rule; `CascadeType` is a Hibernate instruction.

AUTHENTICATION & AUTHORIZATION



HOW AUTHORITIES STORED ?



AUTHORITY vs ROLE



- The names of the authorities/roles are arbitrary in nature and these names can be customized as per the business requirement
- Roles are also represented using the same contract GrantedAuthority in Spring Security.
- When defining a role, its name should start with the ROLE_ prefix. This prefix specifies the difference between a role and an authority.

You can configure the authorization rules to use a different prefix by providing a `GrantedAuthorityDefaults` bean, as shown below:

```
@Bean
static GrantedAuthorityDefaults grantedAuthorityDefaults() {
    //With the given configuration of Spring Security, we can let the framework to generate a random CSRF token
    //which can be sent to UI after successful login. The same token need to be sent by UI for every subsequent
    //requests it is making to backend. For certain paths, we can disable CSRF with the help of
    //ignoringRequestMatchers.
}
```

You need to expose `GrantedAuthorityDefaults` using a static method to ensure that Spring publishes it before initializing Spring Security's method security `@Configuration` classes



In Spring Security the authorities requirements can be configured using the following ways,

hasAuthority() – Accepts a single authority for which the endpoint will be configured and user will be validated against the single authority mentioned. Only users having the same authority configured can invoke the endpoint.

With the given configuration of Spring Security we can let the framework to generate a random CSRF token which can be sent to UI after successful login. The same token need to be sent by UI for every subsequent requests it is making to backend. For certain paths, we can disable CSRF with the help of ignoringRequestMatchers.

hasAnyAuthority() – Accepts multiple authorities for which the endpoint will be configured and user will be validated against the authorities mentioned. Only users having any of the authority configured can invoke the endpoint.

access() – Using Spring Expression Language (SpEL) it provides you unlimited possibilities for configuring authorities which are not possible with the above methods. We can use operators like OR, AND inside access() method.

CONFIGURING AUTHORITIES

You can extract path values from the request, as seen below by using access():

```
http.authorizeHttpRequests((authorize) -> authorize
    .requestMatchers("/profile/{name}").access(new
        WebExpressionAuthorizationManager("#name == authentication.name")))
    With the given configuration of Spring Security, we can let the framework to generate a random CSRF token which can be sent to UI after successful login. The same token need to be sent by UI for every subsequent requests it is making to backend. For certain paths, we can disable CSRF with the help of
```

You can check if the user has multiple roles, as seen below by using access():

```
.requestMatchers("/admin/**").access(allOf(hasAuthority("admin"), hasAuthority("manager")))
```

You can check if the user has multiple roles, as seen below by using access():

```
.requestMatchers("/admin/**").access(new WebExpressionAuthorizationManager(
    "hasAuthority('admin') && hasAuthority('manager')"))
```

CONFIGURING AUTHORITIES

Like shown below, we can configure authority requirements for the APIs/Paths.

```
.authorizeHttpRequests(requests) -> {
    publicPaths.forEach(path ->
        requests.requestMatchers(path).permitAll();
        requests.requestMatchers("/api/v1/admin/**").hasAuthority("ROLE_ADMIN");
        requests.anyRequest().hasAnyAuthority("ROLE_USER", "ROLE_ADMIN");
    )
}
```



In Spring Security the ROLES requirements can be configured using the following ways,

hasRole() – Accepts a single role name for which the endpoint will be configured and user will be validated against the single role mentioned. Only users having the same role configured can invoke the endpoint.

hasAnyRole() – Accepts multiple roles for which the endpoint will be configured and user will be validated against the roles mentioned. Only users having any of the role configured can call the endpoint.

access() – Using Spring Expression Language (SpEL) it provides you unlimited possibilities for configuring roles which are not possible with the above methods. We can use operators like OR, AND inside access() method.

Note :

- ROLE_ prefix only to be used while configuring the role in DB. But when we configure the roles, we do it only by its name.

CONFIGURING ROLES

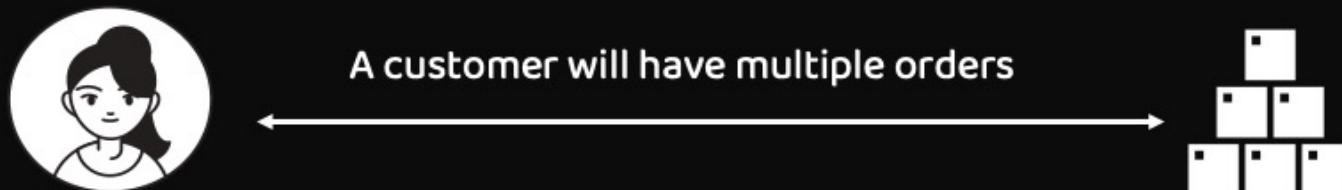
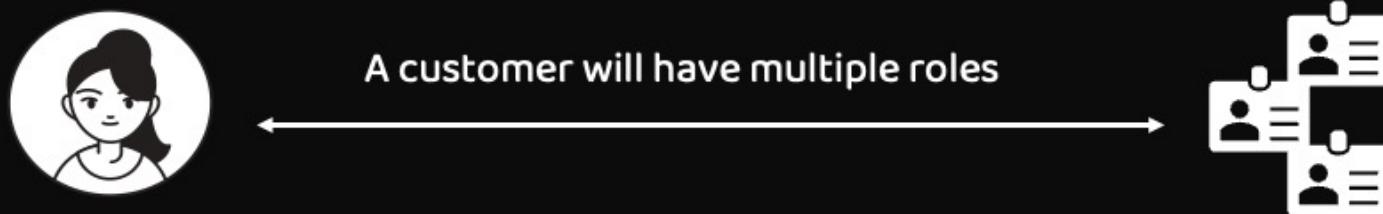
Like shown below, we can configure ROLES requirements for the APIs/Paths.

```
.authorizeHttpRequests((requests) -> {
    publicPaths.forEach(path ->
        requests.requestMatchers(path).permitAll();
        requests.requestMatchers("/api/v1/admin/**").hasRole("ADMIN");
        requests.anyRequest().hasAnyRole("USER", "ADMIN");
    )
})
```

What is a OneToMany Relationship?

@OneToMany is a JPA annotation used to define a one-to-many relationship between two entities. This means: One parent entity can be associated with multiple child entities.

Real-World Examples:



Entities involved:

- Customer (Parent)
- Role (Child)

Customer.java (one side mapping)

```
@Entity
@Table(name = "customers")
public class Customer extends BaseEntity {

    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "customer_id", nullable = false)
    private Set<Role> roles = new LinkedHashSet<>();
}
```

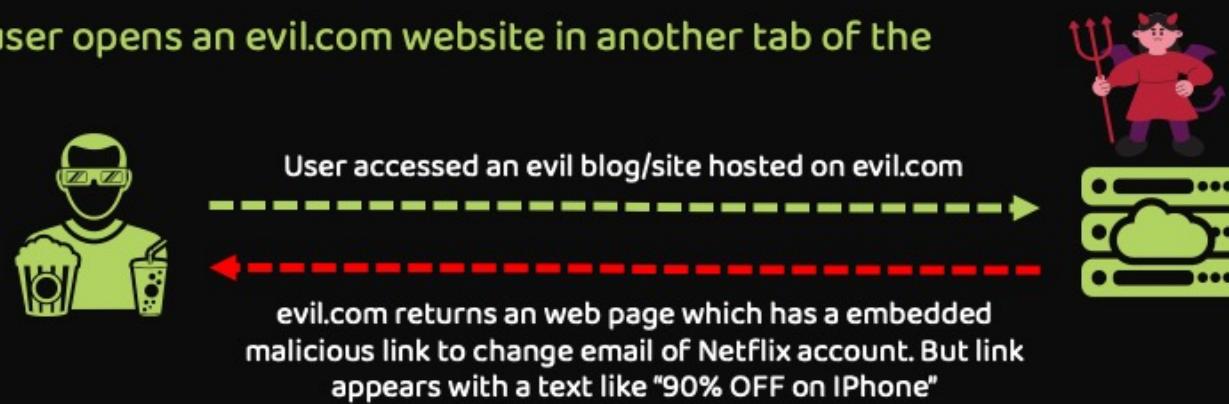
CROSS-SITE REQUEST FORGERY (CSRF)

- A typical Cross-Site Request Forgery (CSRF or XSRF) attack aims to perform an operation in a web application on behalf of a user without their explicit consent. In general, it doesn't directly steal the user's identity, but it exploits the user to carry out an action without their will.
- Consider you are using a website `netflix.com` and the attacker's website `evil.com`

Step 1 : The Netflix user login to `Netflix.com` and the backend server of Netflix will provide a cookie which will store in the browser against the domain name `Netflix.com`



Step 2 : The same Netflix user opens an `evil.com` website in another tab of the browser.



CROSS-SITE REQUEST FORGERY (CSRF)

Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com, the backend server of Netflix.com can't differentiate from where the request came. So here the evil.com forged the request as if it is coming from a Netflix.com UI page.



```
<form action="https://netflix.com/changeEmail"
    method="POST" id="form">
    <input type="hidden" name="email" value="user@evil.com">
</form>

<script>
    document.getElementById('form').submit()
</script>
```

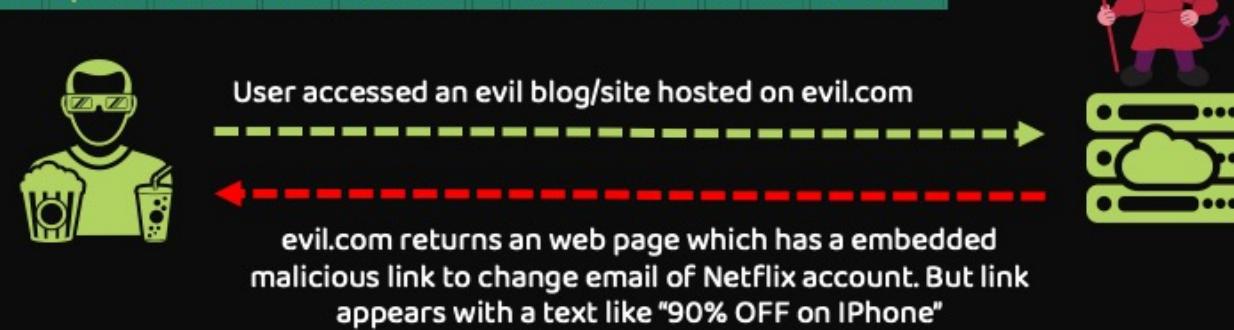
SOLUTION TO CSRF ATTACK

- To defeat a CSRF attack, applications need a way to determine if the HTTP request is legitimately generated via the application's user interface. The best way to achieve this is through a CSRF token. A CSRF token is a secure random token that is used to prevent CSRF attacks. The token needs to be unique per user session and should be of large random value to make it difficult to guess.
- Let's see how this solve CSRF attack by taking the previous Netflix example again,

Step 1: The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com along with a randomly generated unique CSRF token for this particular user session.

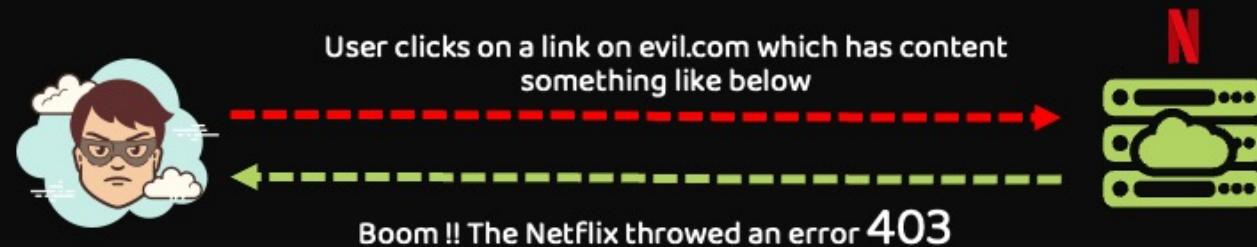


Step 2 : The same Netflix user opens an evil.com website in another tab of the browser.



SOLUTION TO CSRF ATTACK

Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com. This time the Netflix.com backend server expects CSRF token along with the cookie. The CSRF token must be same as initial value generated during login operation



The CSRF token will be used by the application server to verify the legitimacy of the end-user request if it is coming from the same App UI or not. The application server rejects the request if the CSRF token fails to match the test.

DISABLE CSRF PROTECTION

NOT RECOMMENDED
FOR PRODUCTION

By default Spring Security blocks all HTTP POST, PUT, DELETE, PATCH operations with an error of 403, if there is no CSRF solution implemented inside a web application. We can change this default behaviour by disabling the CSRF protection provided by Spring Security.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    return http.csrf(csrfConfig -> csrfConfig.disable())
        .authorizeHttpRequests(requests) -> {
            publicPaths.forEach(path -> requests.requestMatchers(path).permitAll());
            requests.anyRequest().authenticated();
        }).formLogin(withDefaults()).httpBasic(withDefaults()).build();
}
```

CSRF ATTACK SOLUTION

With the given configuration of Spring Security, we can let the framework to generate a random CSRF token which can be provided to UI during initial request. The same token need to be sent by UI for every subsequent requests it is making to backend. For certain paths, we can disable CSRF with the help of ignoringRequestMatchers.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    return http.csrf(csrfConfig -> csrfConfig.csrfTokenRepository
        (CookieCsrfTokenRepository.withHttpOnlyFalse())
        .csrfTokenRequestHandler(new CsrfTokenRequestAttributeHandler())
        .authorizeHttpRequests((requests) -> {
            publicPaths.forEach(path -> requests.requestMatchers(path).permitAll());
            requests.anyRequest().authenticated();
        }).formLogin(withDefaults()).httpBasic(withDefaults()).build());
}
```

Below are the important components that handles CSRF attacks in Spring Security,

CsrfToken - Provides the information/contract about an expected CSRF token.

CsrfTokenRepository – Provides the contract to create, store, and load CSRF tokens. **HttpSessionCsrfTokenRepository** (stores the CsrfToken in the HttpSession), **CookieCsrfTokenRepository** (persists the CSRF token in a cookie named "XSRF-TOKEN" and reads from the header "X-XSRF-TOKEN" Following the conventions of Angular) are the typically used implementation classes.

CsrfTokenRequestHandler – A callback interface that is used to make the CsrfToken created by the CsrfTokenRepository available as a request attribute. Implementations of this interface may choose to perform additional tasks or customize how the token is made available to the application through request attributes. **CsrfTokenRequestAttributeHandler** is the typically used implementation classes.

CsrfFilter – Spring Security in built filter responsible to validate the CSRF token received by client and to decide whether to throw an exception or not.

Implementing Checkout with Stripe – React Overview

Install Required Packages

```
npm install @stripe/react-stripe-js @stripe/stripe-js
```

Load Stripe

- Use `loadStripe()` with publishable key which will return `stripePromise`
- Wrap app in `<Elements stripe={stripePromise}>`

Design Checkout Form

Using Stripe provided elements like `<CardNumberElement />`, `<CardExpiryElement />`, `<CardCvcElement />`

Collect customer Billing Details

Pull logged in customer details from auth-context (e.g., name, email, address)

On Checkout Form Submit:

- Call backend `/create-payment-intent` with amount & currency to get client secret
- Use `stripe.confirmCardPayment(clientSecret, {...})`

On Success:

- Show success message, call `/orders` API to save order data, redirect to `/order-success`



Implementing Checkout with Stripe – SpringBoot Overview

Add Dependencies in pom.xml

```
<dependency>
  <groupId>com.stripe</groupId>
  <artifactId>stripe-java</artifactId>
  <version>XX.X.X</version>
</dependency>
```

Configure Stripe Secret API Key

In application.properties, add the below property,

```
stripe.apiKey=${STRIPE_API_KEY:sk_test_51R....}
```

Stripe Initialization (Config Class)

Use @Configuration class & set Stripe.apiKey in @PostConstruct

REST APIs

Expose /api/v1/payment/create-payment-intent

Use PaymentIntentCreateParams to create payment intent

Return the clientSecret to the frontend



OneToMany & ManyToOne Relationship in Spring Data JPA

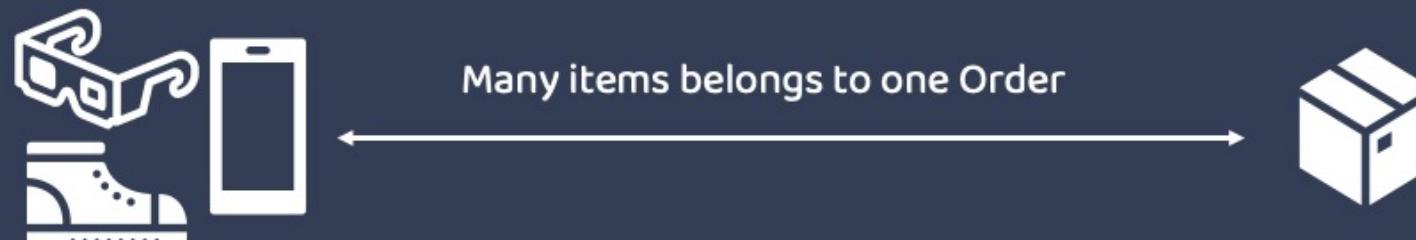
@OneToMany: One parent entity is related to many child entities.

@ManyToOne: Many child entities relate back to one parent entity.

Real-World Examples:

One Order → Many Items

Many Items → One Order



OneToMany & ManyToOne Relationship in Spring Data JPA

Order.java

```
@Entity
@Table(name = "orders")
public class Order extends BaseEntity {

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<OrderItem> orderItems = new ArrayList<>();
}
```

mappedBy = "order": Refers to the order field in OrderItem.

cascade = CascadeType.ALL: Automatically persist/delete child OrderItems when the parent Order changes.

orphanRemoval = true: If an item is removed from the list, it is also deleted from the DB.

OrderItems.java

```
@Entity
@Table(name = "order_items")
public class OrderItem extends BaseEntity {

    @NotNull
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "order_id", nullable = false)
    private Order order;
}
```

@JoinColumn(name = "order_id"): Defines the foreign key column in the order_items table.

Fetch = FetchType.LAZY: Child loads its parent only when needed (better performance).

optional = false: Each order item must be associated with an order.

What is orphanRemoval in JPA?

`orphanRemoval = true` is a JPA configuration used with `@OneToMany` or `@OneToOne` relationships. It automatically deletes a child entity from the database when it is removed from the parent's collection or dereferenced.

Use `orphanRemoval = true` when:

- You want the child entity to have no existence without the parent
- You manage child entities via a Java collection in the parent
- You want to automate delete behavior when removing from collection

Real-world Analogy

- Imagine an Order with a list of OrderItems.
- If you remove an item from the order, it should no longer exist in the database.
- That's what `orphanRemoval = true` does — it treats removed children as orphans and deletes them automatically.



orphanRemoval in JPA

Code example

Order.java

```
@OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)  
private List<OrderItem> orderItems = new ArrayList<>();
```

What Happens with Orphan Removal?

Remove Item from Java List

```
order.getOrderItems().remove(orderItem);
```

JPA will delete that orderItem from the database during flush() or commit().
Without orphanRemoval, the record will still remain in the database, even if it's removed from the list.



How orphanRemoval different from CascadeType.REMOVE?

Feature	orphanRemoval	CascadeType.REMOVE
Trigger	Remove from collection	Remove parent entity
When used	For removing children	For cascading parent delete
Entity Required?	Yes, in Java memory	No, works even without loading

Use both together for full delete lifecycle control.

JPA provides powerful abstractions to interact with databases using standard method names. But sometimes we need custom queries – to join tables, filter specific records, or use native SQL.

JPA offers 3 main ways:

1. `@Query`
2. `@NamedQuery`
3. `@NamedNativeQuery`

JPA supports two types of queries:

JPQL (Java Persistence Query Language): Looks like SQL but works with entity names and fields, not table/column names.

Native SQL: Regular SQL queries that work directly with database tables.



@Query – Inline JPQL (Most Common Way)

Use **@Query** annotation on repository methods to define JPQL (Java Persistence Query Language).

Easy to write and maintain.

Supports named and positional parameters.

```
public interface OrderRepository extends JpaRepository<Order, Long> {

    @Query("SELECT o FROM Order o WHERE o.customer = :customer ORDER BY o.createdAt DESC")
    List<Order> findOrdersByCustomer(@Param("customer") Customer customer);

    @Query("SELECT o FROM Order o WHERE o.orderStatus = ?1")
    List<Order> findOrdersByStatus(String orderStatus);

}
```

Explanation:

- **SELECT o FROM Order o:** Fetches all orders.
- **WHERE o.customer = :customer:** Filters by the given customer.
- **ORDER BY o.createdAt DESC:** Orders results by createdAt descending.
- **@Param("customer"):** Binds the method parameter to the JPQL named parameter.
- **?1 refers to the first method parameter (String orderStatus).**

What Are Native Queries?

- Native Queries are SQL queries written directly for the underlying database.
- Unlike JPQL, they use table names and column names (not entity names or fields).
- Useful when JPQL doesn't support certain database-specific features or performance tuning is needed.

```
public interface OrderRepository extends JpaRepository<Order, Long> {  
  
    @Query(value = "SELECT * FROM orders WHERE customer_id = :customerId ORDER BY created_at DESC",  
          nativeQuery = true)  
    List<Order> findOrdersByCustomerWithNativeQuery(@Param("customerId") Long customerId);  
  
    @Query(value = "SELECT * FROM orders WHERE order_status = ?1", nativeQuery = true)  
    List<Order> findOrdersByStatusWithNativeQuery(String orderStatus);  
}
```

Explanation:

- For native queries, you must use database column names, not entity field names.
- Since native queries don't automatically resolve entity associations like Customer, you should pass the customerId (a Long) instead of the full Customer object.

In Spring Data JPA, you can use @Query with DML (Data Manipulation Language) operations like UPDATE and DELETE. However, there are three key requirements:

- Use @Modifying annotation along with @Query
- The method must return void or int (number of affected rows)
- Wrap it with @Transactional

Example: Update Order Status by Order ID

```
@Transactional
@Modifying
@Query("UPDATE Order o SET o.orderStatus=:orderStatus,o.updatedAt=CURRENT_TIMESTAMP,
o.updatedBy=:updatedBy WHERE o.orderId=:orderId")
int updateOrderStatus(@Param("orderId") Long orderId, @Param("status") String orderStatus);
```

Important Notes:

- @Modifying tells Spring Data that the query will change data (not a SELECT).
- You cannot use a method like Order updateOrderStatus(...) to return the updated entity directly – instead, fetch it again if needed.
- Works with both JPQL and native queries (set nativeQuery = true).
- Use @Transactional at the method or service level
 - ✓ On whichever method/class we declare @Transactional the boundary of transaction starts and boundary ends when method execution completes. Lets say you are updating entity1 and entity2. Now while saving entity2 an exception occur, then as entity1 comes in same transaction so entity1 will be rollback with entity2. Any exception will result in rollback of all JPA transactions with DB.

@NamedQuery – Predefined Query on Entity

Defined at the entity level using @NamedQuery annotation. Good for reusable queries, loaded at app startup.

✓ Pros: Precompiled and cached at startup, Centralized queries.

⚠ Cons: Requires a matching repository method name

```
@Entity
@Table(name = "contacts")
@NamedQuery(name = "Contact.findByStatus", query = "SELECT c FROM Contact c WHERE c.status =:status")
public class Contact extends BaseEntity {
    // ... existing fields ...
}
```

```
public interface ContactRepository extends JpaRepository<Contact, Long> {

    // This automatically uses the NamedQuery
    List<Contact> findByStatus(String status);
}
```

- @NamedQuery is a JPQL query and uses entity and field names, not table/column names.
- The method name `findByStatus` matches the NamedQuery name: `Contact.findByStatus`, so Spring Data will resolve it automatically.
- You don't need `@Query` on the repository method.

@NamedNativeQuery – Raw SQL Query on Entity

Defined at the entity level but uses raw SQL (not JPQL). Useful for complex queries or database-specific features.

✓ Pros: Full SQL power (joins, indexes, vendor-specific syntax).

⚠ Cons: Tied to database structure, Risk of breaking on schema changes.

```
@Entity
@Table(name = "contacts")
@NamedNativeQuery(name = "Contact.findByStatusWithNativeQuery", query = "SELECT * FROM contacts WHERE
    status = :status", resultClass = Contact.class)
public class Contact extends BaseEntity {
    // ... existing fields ...
}
```

```
public interface ContactRepository extends JpaRepository<Contact, Long> {

    // This automatically uses the NamedNativeQuery
    List<Contact> findByStatusWithNativeQuery (String status);
}
```

@NamedQueries and @NamedNativeQueries

We can create multiple named queries and named native queries using the annotations **@NamedQueries** and **@NamedNativeQueries**. Below is the syntax of the same,

```
@NamedQueries({  
    @NamedQuery(name = "one", query = "?"),  
    @NamedQuery(name = "two", query = "?")  
})
```

```
@NamedNativeQueries({  
    @NamedNativeQuery(name = "one", query = "?", resultClass = ?),  
    @NamedNativeQuery(name = "two", query = "?", , resultClass = ?)  
})
```

What is @Transactional?

@Transactional is an annotation in Spring used to manage transactions in a declarative way.

Purpose:

It ensures that a series of database operations either all succeed or all fail – no partial updates.

Basic Syntax

```
@Service
public class BankService {

    @Transactional
    public void transferMoney(Long fromId, Long toId, BigDecimal amount) {
        debit(fromId, amount);
        credit(toId, amount);
    }
}
```



Spring will automatically start a transaction, and:

- Commit it if no exception occurs
- Roll it back if an exception is thrown

Rollback Rules

By default, rollback happens for `RuntimeException` or `Error`

Not for checked exceptions (like `IOException`) unless you configure it:

```
@Transactional(rollbackFor = IOException.class)
```

Read-Only Transactions

```
@Transactional(readOnly = true)
public List<Product> getAllProducts() {
    return productRepository.findAll();
}
```

Why use it?

Optimizes performance by telling Hibernate/Spring:

👉 "Don't track changes, I'm just reading"

Where @Transactional will not work ?

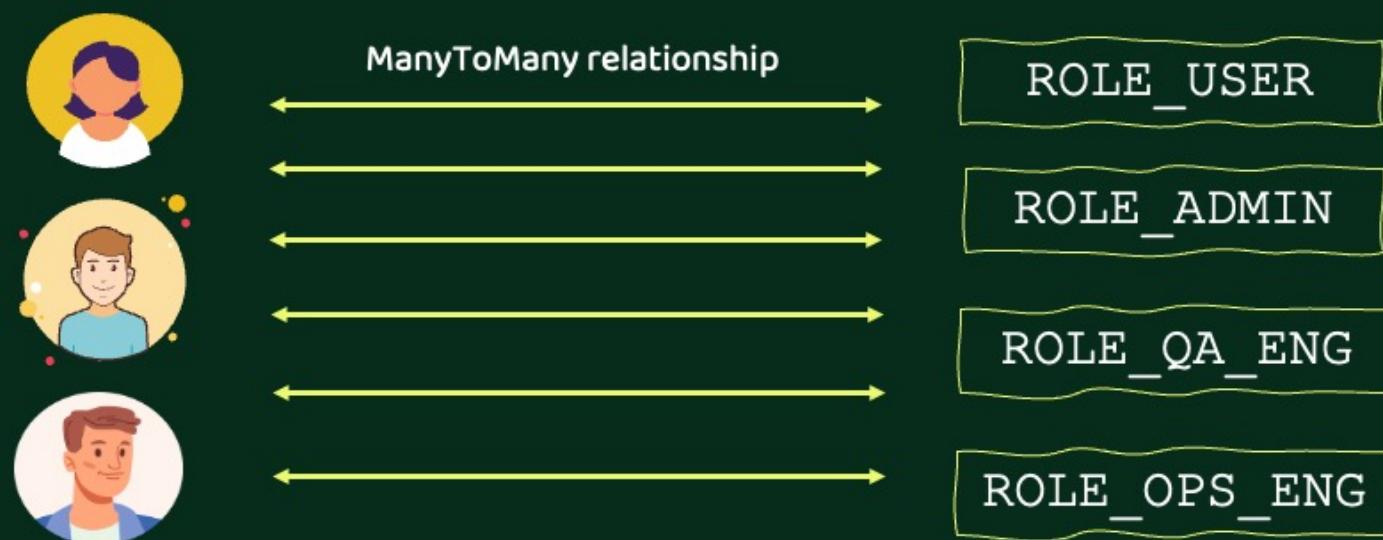
- If the method is private or final, Spring can't proxy it
- Transactions only work on public methods of Spring-managed beans

What is @ManyToMany?

Used when both sides of the relationship can have multiple references to each other.

Example:

- A Customer can have multiple Roles
- A Role (like ADMIN, USER) can be assigned to multiple Customers



@ManyToMany Relationship in JPA

Why do we need a third table in a Many-to-Many relationship?

When two entities share a many-to-many relationship – such as customer and role – it's best to use a third (join) table to represent the association.

For example, if a customer has multiple roles, using a separate join table prevents duplication of data in the customers or roles tables. Without it, we'd need to store repetitive or nested data.



@ManyToMany Relationship in JPA

Why Not Use @OneToMany?

@OneToMany means a Role belongs to only one Customer → not reusable

@ManyToMany allows shared roles – cleaner and more flexible design

Database Design

customers table – stores customer info

roles table – stores unique role names

customer_roles (Join Table) – connects both with foreign keys

Best Practices

Use a Set<> to avoid duplicates.

Avoid CascadeType.REMOVE in @ManyToMany to prevent accidental deletions.

Keep role names unique using a constraint.

@ManyToMany Relationship in JPA

Customer.java

```
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(
    name = "customer_roles",
    joinColumns = @JoinColumn(name = "customer_id"),
    inverseJoinColumns = @JoinColumn(name = "role_id"))
private Set<Role> roles = new LinkedHashSet<>();
```

Role.java

```
@ManyToMany(mappedBy = "roles")
private Set<Customer> customers = new LinkedHashSet<>();
```

In any bidirectional relationship, only one side is the owner.

- The owning side defines the `@JoinTable`.
- The inverse side uses `mappedBy` to point back to the owning side's field.

Why Use Caching?

Avoid repeated database calls for static or rarely changing data.

Improve application performance by reducing load on backend services.

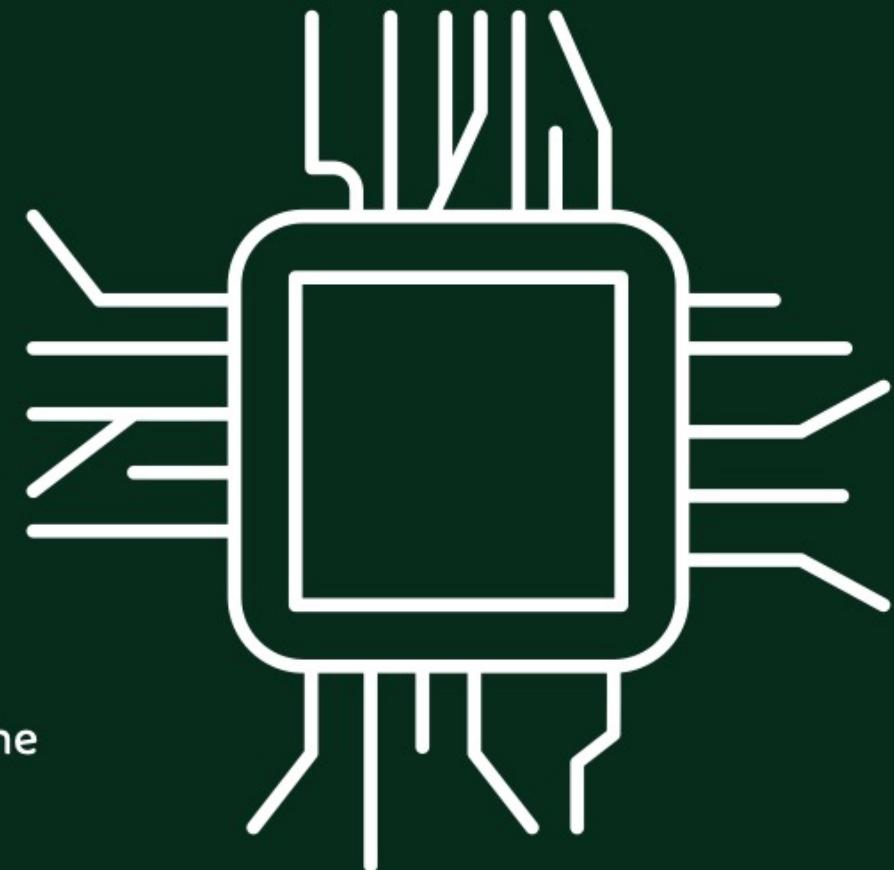
Perfect for reference tables like roles, countries, categories, etc.

Scenario Example: Static roles Table

We often need to fetch role data like "ROLE_USER" from the database:

```
roleRepository.findByName("ROLE_USER")
    .ifPresent(role -> customer.setRoles(Set.of(role)));
```

These roles are static. Instead of hitting the DB every time, we can cache the result.



Enable Caching in Spring Boot

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

```
@EnableCaching
@SpringBootApplication
public class EazystoreApplication { ... }
```

Common Caching Annotations

Annotation	Purpose	Example use case
@Cacheable	Reads from cache if present	Fetch role by name
@CachePut	Executes method & updates the cache	Update role and refresh cache
@CacheEvict	Removes entry from cache	Delete role or clear cache

Example code of using caching related annotations

```
public interface RoleRepository extends JpaRepository<Role, Long> {

    // ✅ Caches the role by name on first fetch
    @Cacheable("roles")
    Optional<Role> findByName(String name);

    // 🔄 Updates cache when role is saved/updated
    @CachePut(value = "roles", key = "#role.name")
    Role save(Role role);

    // ❌ Evicts role from cache when deleted by name
    @CacheEvict(value = "roles", key = "#name")
    void deleteByName(String name);

    // ⚠️ Optional: Clear all role cache entries
    @CacheEvict(value = "roles", allEntries = true)
    void deleteAll(); // Only if appropriate
}
```

Tips for Effective Caching

- Use `@Cacheable` on read-only or static lookup methods.
- Use `@CachePut` when updating records and want to sync cache.
- Use `@CacheEvict` when deleting or modifying the cached data.
- Avoid using `@Cacheable` on write/update operations

Spring Caching with Auto-Expiry

To control expiry (TTL), you'll need a real cache provider like Caffeine or Redis, because the default in-memory cache (`ConcurrentMapCacheManager`) doesn't support TTL.

Use Caffeine Cache (lightweight, in-memory, fast)

What is Caffeine?

- A high-performance, in-memory caching library by the makers of Guava
- Designed as a modern replacement for `ConcurrentHashMap` or `EhCache`
- Ideal for caching read-heavy, frequently accessed data

Add below maven dependency

```
<dependency>
    <groupId>com.github.ben-manes.caffeine</groupId>
    <artifactId>caffeine</artifactId>
</dependency>
```

Spring Caching with TTL(Time-To-Live) Configuration

Example code of TTL related configs using Caffeine

```
@Configuration
public class CaffeineCacheConfig {

    @Bean
    public CacheManager caffeineCacheManager() {
        CaffeineCache productsCache = new CaffeineCache("products",
            Caffeine.newBuilder()
                .expireAfterWrite(30, TimeUnit.MINUTES)
                .maximumSize(1000)
                .build());

        CaffeineCache rolesCache = new CaffeineCache("roles",
            Caffeine.newBuilder()
                .expireAfterWrite(1, TimeUnit.DAYS)
                .maximumSize(1)
                .build());

        SimpleCacheManager manager = new SimpleCacheManager();
        manager.setCaches(Arrays.asList(productsCache, rolesCache));
        return manager;
    }
}
```

What happens with CacheManager configurations ?

Cache name	TTL Durations	Max Entries
products	30 Minutes	1000
roles	1 Day	1

- Entries expire after the TTL duration from their last write
- Keeps memory usage in control
- Automatically refreshes stale data on the next call

HOW TO READ PROPERTIES IN SPRINGBOOT APPS

In Spring Boot, there are multiple approaches to reading properties. Below are the most commonly used approaches,

Using @Value Annotation



You can use the `@Value` annotation to directly inject property values into your beans. This approach is suitable for injecting individual properties into specific fields.

For example:

```
@Value("${property.name}")
private String propertyName;
```

Using Environment



The `Environment` interface provides methods to access properties from the application's environment. You can autowire the `Environment` bean and use its methods to retrieve property values. This approach is more flexible and allows accessing properties programmatically. For example:

```
@Autowired
private Environment env;
```

```
public void getProperty() {
```

```
    String propertyName =
        env.getProperty("property.name");
}
```

Using @ConfigurationProperties



Recommended approach as it avoids hard coding the property keys

The `@ConfigurationProperties` annotation enables binding of entire groups of properties to a bean. You define a configuration class with annotated fields matching the properties, and Spring Boot automatically maps the properties to the corresponding fields.

```
@ConfigurationProperties("prefix")
public class MyConfig {
    private String property;
```

```
    // getters and setters
}
```

`@EnableConfigurationProperties` needs to mention on top of the SpringBoot main class, to enable the above feature

Reading Properties with @PropertySource

What is @PropertySource?

@PropertySource is an annotation in Spring that tells the application to load properties from an external file. This helps you keep configurations separate from your code.

Why use @PropertySource?

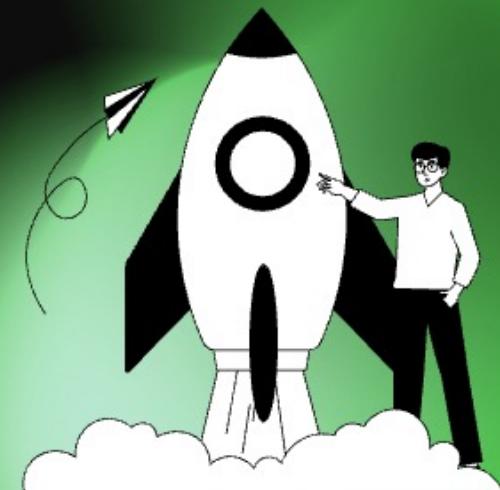
To load custom properties files other than application.properties
To organize configuration logically

How to use it?

You add @PropertySource in any @Configuration class:

```
@Configuration  
@PropertySource("classpath:custom-config.properties")  
public class AppConfig {  
}
```

👉 This will load the file custom-config.properties from src/main/resources



Reading Properties with @PropertySource

🔑 Accessing properties with @Value

You can access the values in your Spring beans:

```
@Value("${app.title}")
private String appTitle;
```

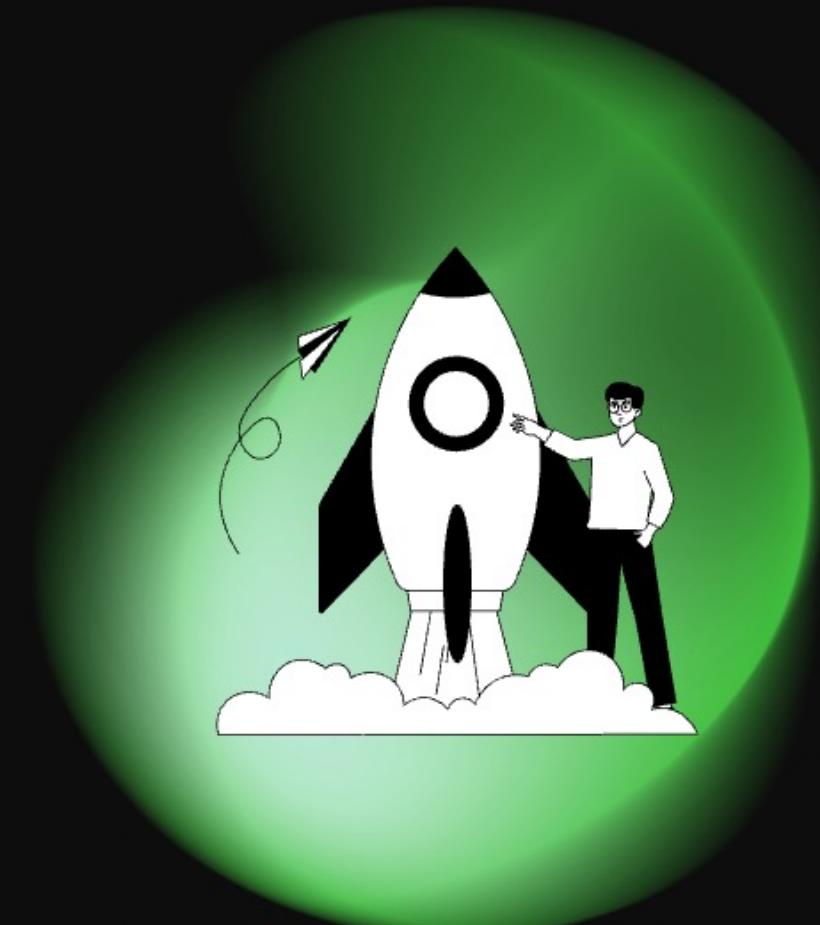
If custom-config.properties has:

```
app.title=My Spring App
```

appTitle will have the value "My Spring App".

👉 You can load multiple files as well using the below syntax:

```
@PropertySource({ "classpath:db.properties",
    "classpath:app.properties" })
```



Reading Properties with @PropertySource

ignoreResourceNotFound in @PropertySource

By default, Spring throws an error if the specified properties file does not exist.

👉 Use `ignoreResourceNotFound = true` to avoid this error.

This makes your application more flexible if the file is optional.

```
@Configuration
@PropertySource(
    value = "classpath:optional-config.properties",
    ignoreResourceNotFound = true
)
public class AppConfig {
```

⌚ When to use?

- When the file is optional
- For external properties files which may not exist in all environments
- To avoid application startup failures



EXTERNALIZING PROPERTIES

Spring Boot allows you to externalize configuration/properties, so the same code works in different environments.

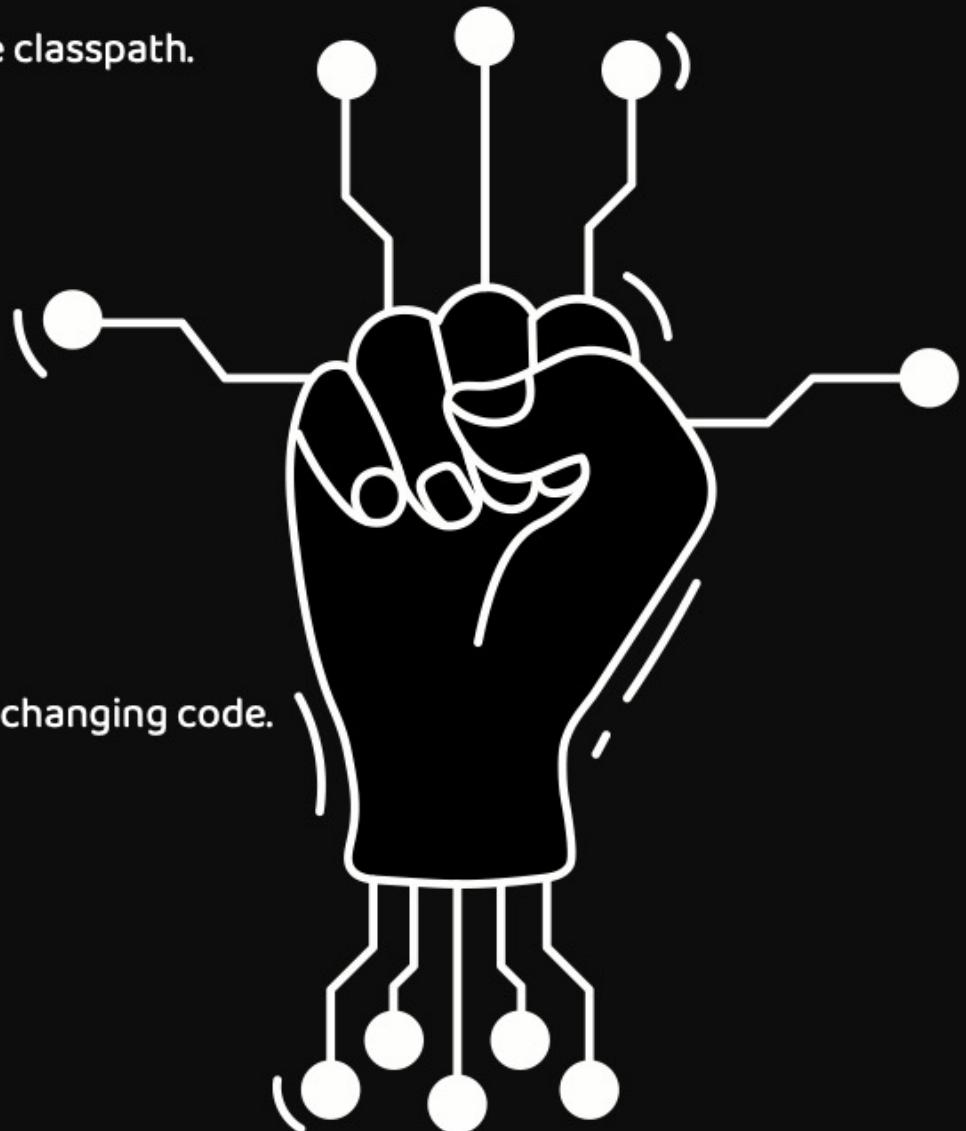
By default, Spring Boot looks for application.properties or application.yml in the classpath.

Spring Boot follows a specific order of precedence for configurations.

Later sources override earlier ones:

- Properties in application.properties or application.yml
- Operating system environment variables
- Java system properties (System.getProperties())
- JNDI attributes (java:comp/env)
- ServletContext init parameters
- ServletConfig init parameters
- Command-line arguments

This order makes it easy to override values for different environments without changing code.



Externalize configurations using Environment variables

Environment variables are popular for external configuration because they work across all operating systems. Java can read them using `System.getenv()`.

To map an environment variable to a Spring property:

- Change letters to uppercase
- Replace dots (.) and dashes (-) with underscores (_)

Example:

`BUILD_VERSION` → maps to Spring property `build.version`

This automatic conversion is called relaxed binding in Spring Boot.

Windows

```
env:SPRING_JPA_SHOW_SQL=false; java -jar accounts-service-0.0.1-SNAPSHOT.jar
```

Linux based OS

```
SPRING_JPA_SHOW_SQL=false java -jar accounts-service-0.0.1-SNAPSHOT.jar
```



Externalize configurations using command-line arguments

Spring Boot converts command-line arguments into key/value pairs and adds them to the Environment object.

In production, these arguments have the highest priority.

You can override configuration by passing command-line arguments when running your JAR.

```
java -jar easystore-0.0.1-SNAPSHOT.jar --spring.jpa.show-sql=false
```

The command-line argument follows the same naming convention as the corresponding Spring property, with the familiar -- prefix for CLI arguments.



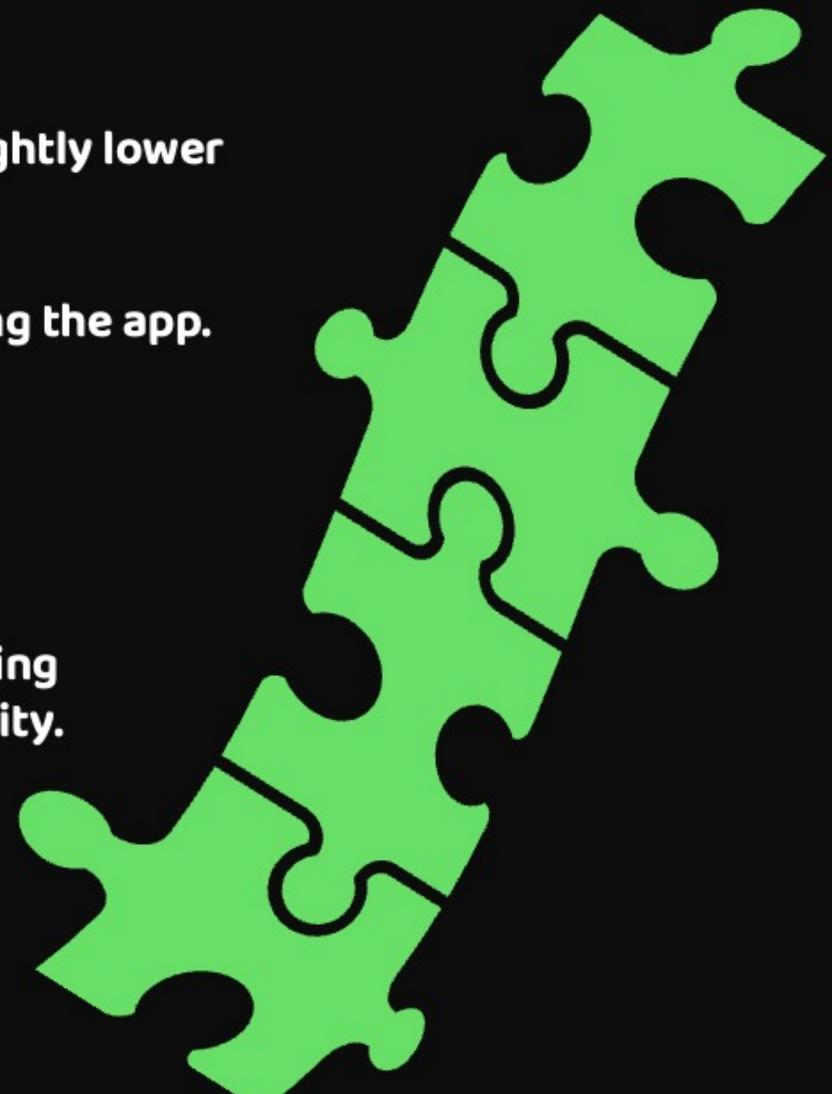
Externalize configurations using JVM System properties

JVM system properties can also override Spring properties, but have slightly lower priority than command-line arguments.

You can pass JVM properties using `-Dproperty.name=value` when starting the app. This helps you change settings without rebuilding the JAR.

```
java -Dspring.jpa.show-sql=false -jar easystore-0.0.1-SNAPSHOT.jar
```

If both a JVM property and a command-line argument are provided, Spring Boot uses the command-line argument value, as it has the highest priority.

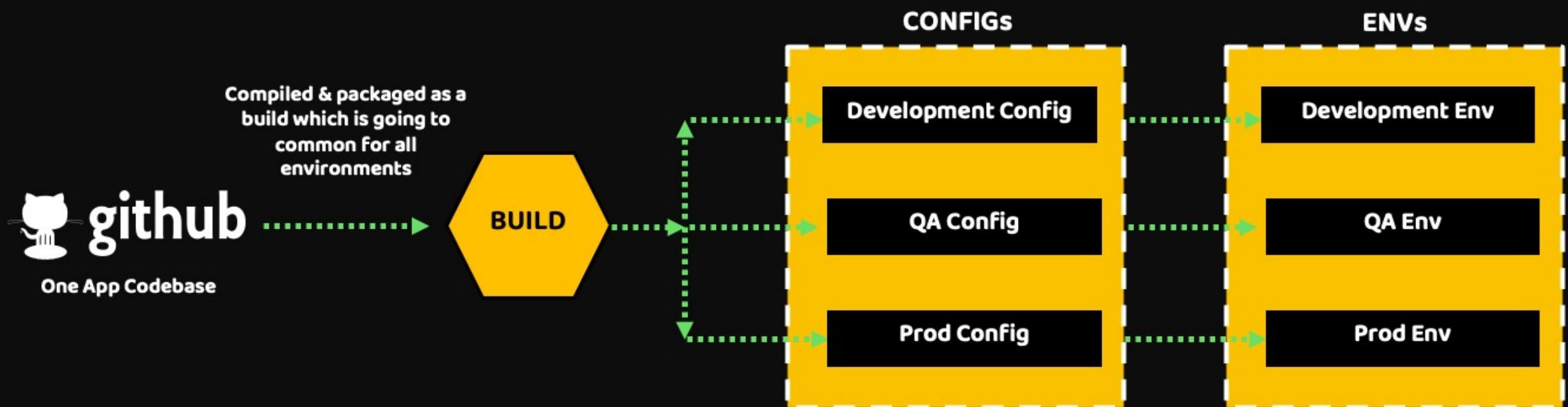


Environment-Specific Configuration

Imagine a backend app has following needs across multiple environments:

- **Development:** Use H2 database, enable debug logging, and run on port 8080.
- **Production:** Use PostgreSQL, disable debug logging, and run on port 80.
- **Testing:** Use a mock email service and a test database.

Changing the `application.properties` file manually before the deployment of the app into a environment is a cumbersome and error prone process. We need a better and automated approach



Spring Boot Profiles

What Are Spring Boot Profiles?

Spring Boot profiles allow us to create multiple configurations for different environments in a single application.

👉 Example environments:

- Development (dev)
- Testing (test)
- Staging (staging)
- Production (prod)

You can define different properties, beans, or behaviors for each environment.

Why Do We Need Profiles?

Imagine an application running:

- On a developer's laptop → uses local database
- On QA server → uses test database
- On production → uses secure production database

👉 The same code but different configurations

👉 Profiles avoid manual code changes between environments



Spring Boot Profiles

Analogy - Clothing for Seasons

Think of Spring profiles like seasonal clothing:

Summer profile → Light clothes

Winter profile → Warm clothes

Rainy profile → Raincoat and boots

Your body (application) is same,

  clothing (profile configuration) changes depending on weather (environment).

Similarly:

Your application is same,

Profiles change configuration for the environment it runs in.

What Can You Change with Profiles?

👉 You can customize almost anything!

- application.properties files
- Database connection
- API URLs
- Logging levels
- Bean creation (conditional beans)



Spring Boot Profiles

How to Define Profiles?

Define profile-specific properties in files like `application-<profile>.yml` or `application-<profile>.properties`

For examples,

`application-prod.properties` for prod env/profile
`application-qa.properties` for qa env/profile

The default profile is always active. Spring Boot loads all properties in `application.properties` into the default profile.

We can activate a specific profile using `spring.profiles.active` property either by directly mentioning inside the corresponding file or by providing externally using CLI, Env variables etc.

Activate profiles via:

Command-line: `--spring.profiles.active=prod`

Environment variable: `SPRING_PROFILES_ACTIVE=prod`

Application properties: `spring.profiles.active=prod`



Conditional Bean Creation in Spring Boot

What is Conditional Bean Creation?

Sometimes you want to create a bean only if a certain condition is true. Spring Boot provides powerful annotations to create beans conditionally:

This is useful for:

- Switching configurations
- Enabling features for certain environments
- Avoiding unnecessary beans in production

@ConditionalOnProperty

Create a bean ***only if a property exists or has a specific value.**

```
@Bean  
@ConditionalOnProperty(name = "feature.enabled",  
                      havingValue = "true")  
public MyService myService() {  
    return new MyService();  
}
```

If you set **feature.enabled=true** in application.properties, the bean will be created.



Conditional Bean Creation in Spring Boot

@Profile

Create a bean only for a specific profile.

```
@Profile("dev")
@Bean
public DataSource devDataSource() {
    return new DevDataSource();
}
```

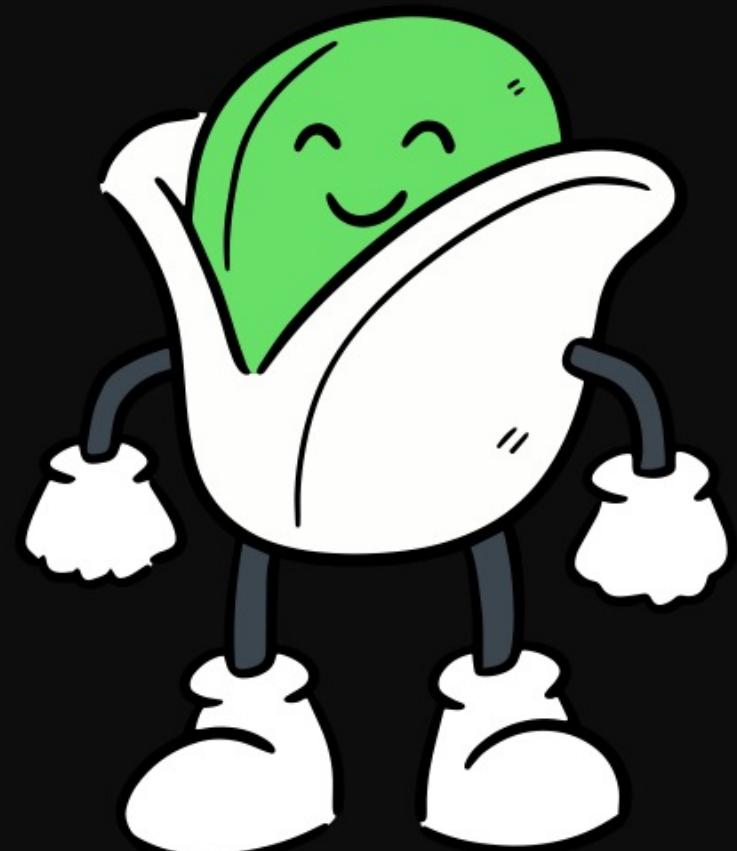
👉 This bean is active only when `spring.profiles.active=dev`

@ConditionalOnMissingBean

Create a bean only if the bean does not already exist.

```
@Bean
@ConditionalOnMissingBean
public MyService myService() {
    return new MyService();
}
```

👉 Useful for providing default beans that users can override.



Conditional Bean Creation in Spring Boot

@ConditionalOnClass

Create a bean only if a specific class exists in the classpath.

```
@Bean  
@ConditionalOnClass(name = "com.example.SomeLibrary")  
public MyLibraryBean myLibraryBean() {  
    return new MyLibraryBean();  
}
```

👉 Great for auto-configurations.

⌚ Benefits of Conditional Beans

- Clean & flexible configuration
- Activate features only when needed
- Reduce memory and startup time
- Avoid duplicate beans

The above concepts applicable for stereotype annotations as well.



What is Redux?

Redux is a state management library that helps you manage global data in your app. Instead of passing data through many components using props drilling, Redux gives you one place to store all shared data.

Think of Redux like a central warehouse where all components go to get or update data.



What is Redux Toolkit (RTK)?

In the recent times, RTK is the official way to use Redux.

- Reduces boilerplate
- Makes Redux setup faster
- Built-in tools like `createSlice`, `configureStore`

You don't need to write action types or switch statements manually!

Feature	React Context API	Redux
Built-in	Yes	No (Separate library)
Global State	Yes	Yes
Setup Complexity	Simple	Requires setup (but RTK helps)
Performance	May re-render entire tree	Efficient (with selectors)
Async Support	Manual work	Built-in via middleware
Dev/Debugging Tools	Limited	Powerful

React Context is great for:

- Less frequent state updates like Theme toggling, User login status, Language preference
- Small, Medium apps with simple state

Redux is better for:

- High frequent state updates like Shopping cart, Notifications, API calls and loading states
- Larger apps with complex state data

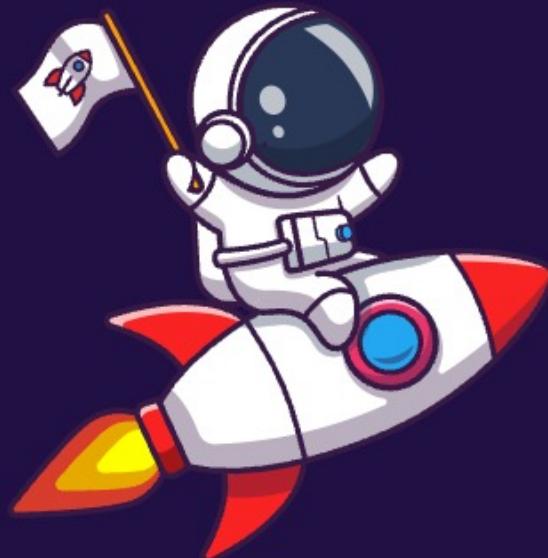


How about Performance ?

⚠️ React Context can cause unnecessary re-renders

Example: Updating a user's name in context might re-render all components using the same context, even if they don't use the name.

✓ Redux uses useSelector() to only re-render components that need the updated data.



Can We Use Both?

Absolutely!

👉 Many apps use both:

Context for simple global flags (e.g., theme)
Redux for complex app-wide data

What is a Reducer? - A reducer is just a function that takes the current state and an action, then returns a new state.

```
(currentState, action) => newState
```

useReducer in React - React gives you useReducer to manage local component state using this pattern:

```
const [state, dispatch] = useReducer(reducerFn, initialState);
```

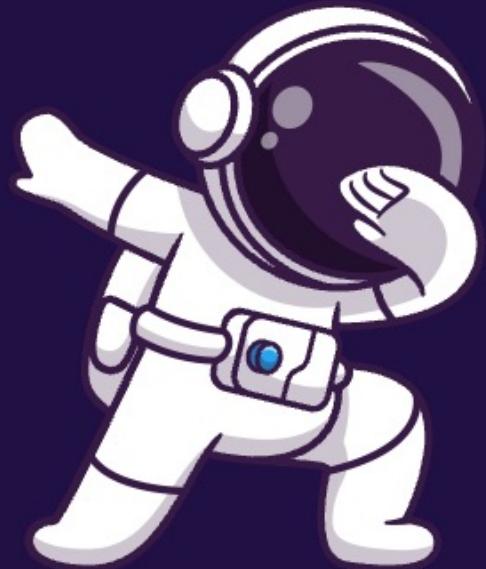
You dispatch actions like: `dispatch({ type: "INCREMENT" })`

Redux (Global State)

Redux applies the same idea—but globally for your whole app.

In Redux:

- You define a reducer
- You dispatch actions
- Redux updates the global state and gives the new state to your app



Redux (2015) came before useReducer (2018). useReducer was inspired by Redux's reducer pattern. Think of useReducer as a lightweight Redux for one component.

Building blocks of Redux

Store: The central place for all state.

Slice: A piece of state (e.g., cart) with actions, reducers, selectors.

Actions: Messages to update state.

Reducers: Logic to change state.

Selectors: Tools to read state.



Analogy

Redux is a librarian:

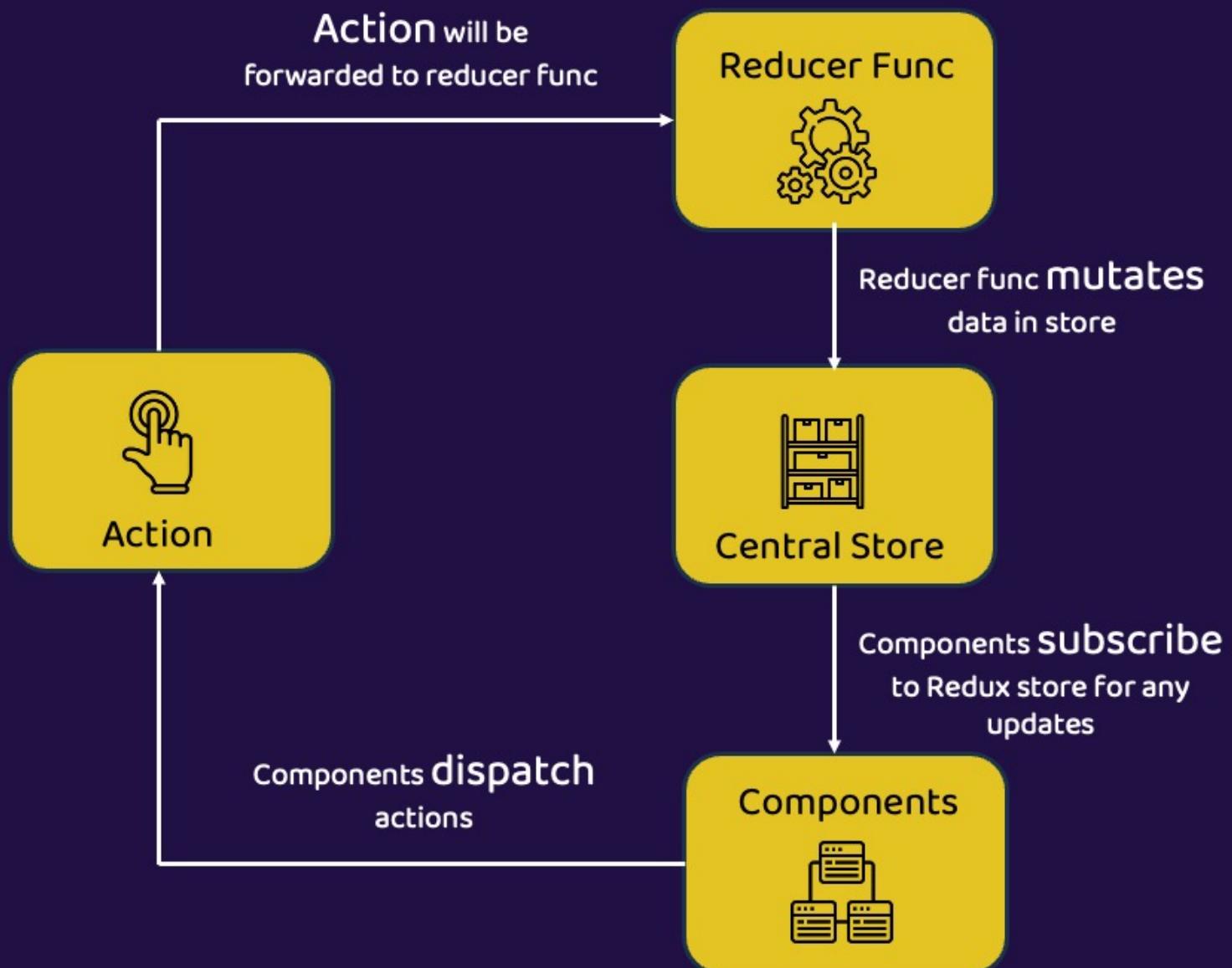
Store = library (holds all data).

Slice = a particular section in library

Actions = requests (e.g., "borrow a book").

Reducers = rules for updating the library.

Components = readers who ask for books.



1. `createSlice()` - Creates a slice of state with initial state, reducers, and actions.

Analogy: It's like creating a mini module. 📦 State + 🛡 Functions = 1 file (slice)

2. `configureStore()` - Sets up the global Redux store where all your slices live.

Analogy: Think of this as the main register that combines all mini modules (slices).

3. `Provider` - A React wrapper that connects Redux to your app.

Analogy: Like connecting all the pipes in your house to one central water tank.

4. `useSelector()` - To read data from the store.

Analogy: It's like viewing the state from a window 📄 — no changes, just read.

5. `useDispatch()` - To send an action (e.g., increase counter, add product) from React components

Analogy: Like pressing a button that triggers a machine to update something.



counter-slice.js

```
import { createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
  name: "counter",
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1 },
    decrement: (state) => { state.value -= 1 },
  }
});

export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

The `createSlice` function returns an object with the following main properties:

```
{
  name,        // the name of the slice
  reducer,     // the generated reducer function
  actions,     // an object containing action creators
}
```

Think of `reducers` as the recipe you write (`createSlice({... reducers: {...}})`), and `reducer` as the final dish (the function) that Redux uses to update the store.



store.js - Tells Redux how many slices we have and combines them into one store.

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counter-slice";

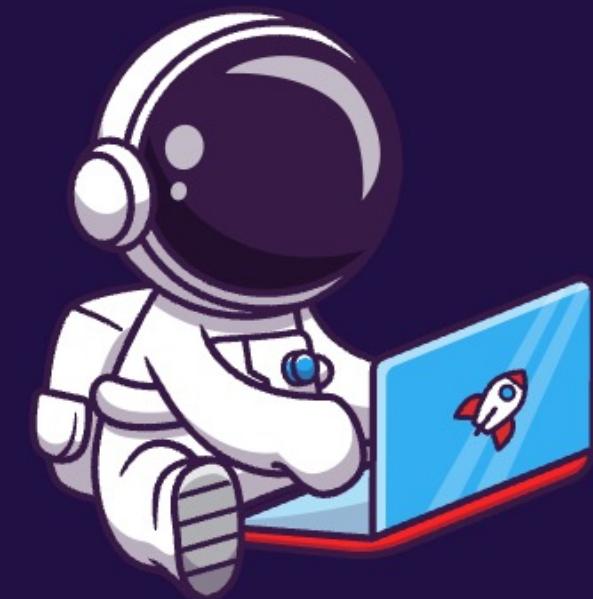
const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});
```

You don't have to use the same name ("counter") as the one you provided in `createSlice({ name: "counter", ... })` when registering it in `configureStore`.

The name you provide in `createSlice` is used primarily for: Generating action type strings like "counter/increment" or "counter/decrement" (which helps with debugging and logging).

But when you register the reducer in the `configureStore`, the key in the reducer object determines how your state will be structured in the Redux store, like this:

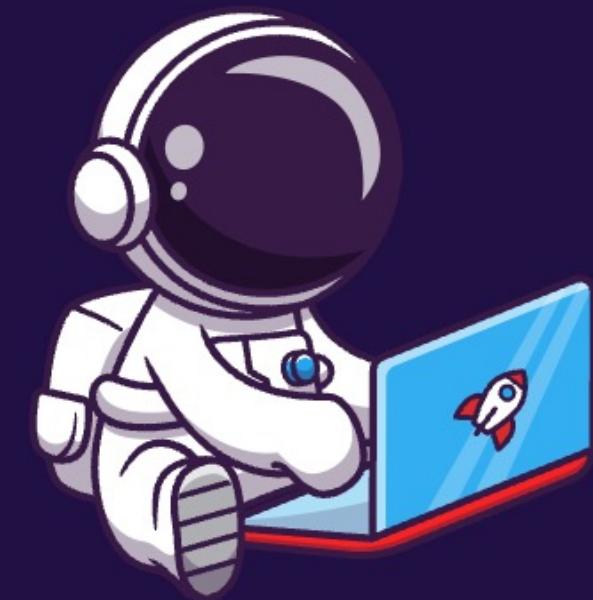
```
{
  counter: {
    value: 0
  }
}
```



main.jsx - Makes Redux available to every component in the app

```
import { Provider } from "react-redux";
import store from "./store";

<Provider store={store}>
  <App />
</Provider>
```

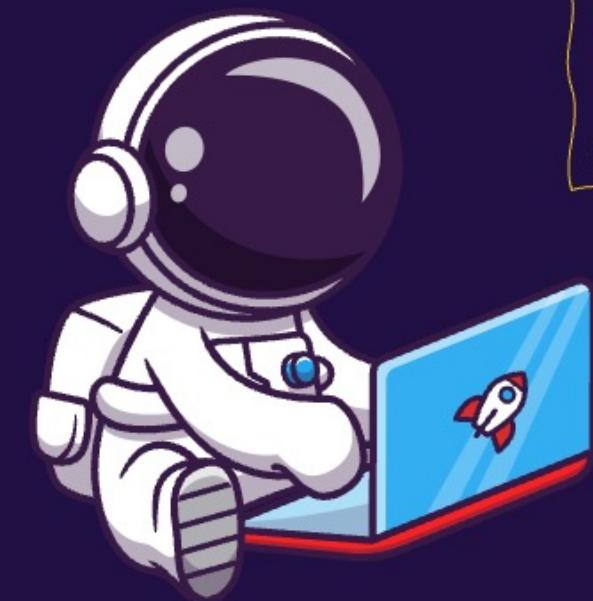


Counter.jsx - Makes Redux available to every component in the app

```
import { useSelector, useDispatch } from "react-redux";
import { increment, decrement } from "./counter-slice";

function Counter() {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();

  return (
    <>
      <h2>Counter: {count}</h2>
      <button onClick={() => dispatch(increment())}>+ Add</button>
      <button onClick={() => dispatch(decrement())}>- Subtract</button>
    </>
  );
}
```



How to Pass Payload in Redux Toolkit

Define the reducer to accept `action.payload`

```
const counterSlice = createSlice({
  name: "counter",
  initialState: { value: 0 },
  reducers: {
    incrementByAmount: (state, action) => {
      // action.payload contains the value passed from the component
      state.value += action.payload;
    },
  },
});

export const { incrementByAmount } = counterSlice.actions;
```

Dispatch it from your component with payload



```
import { useDispatch } from "react-redux";
import { incrementByAmount } from "../store/counter-slice";

const dispatch = useDispatch();

<button onClick={() => dispatch(incrementByAmount(10))}>
  Add 10
</button>
```

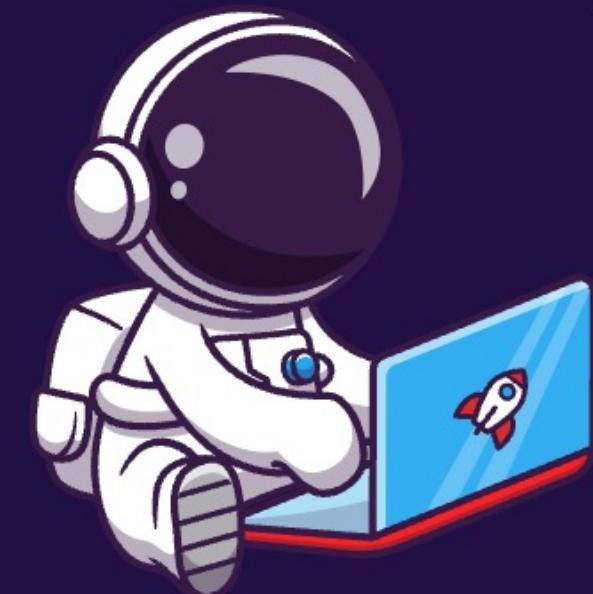
Example: Increment Counter

React Context (Immutable)

```
function reducer(state, action) {  
  switch (action.type) {  
    case "INCREMENT":  
      return { ...state, value: state.value + 1 }; // must copy  
    }  
}
```

Redux Toolkit (Looks Mutable)

```
increment: (state) => {  
  state.value += 1; // looks like mutation ✓  
}
```



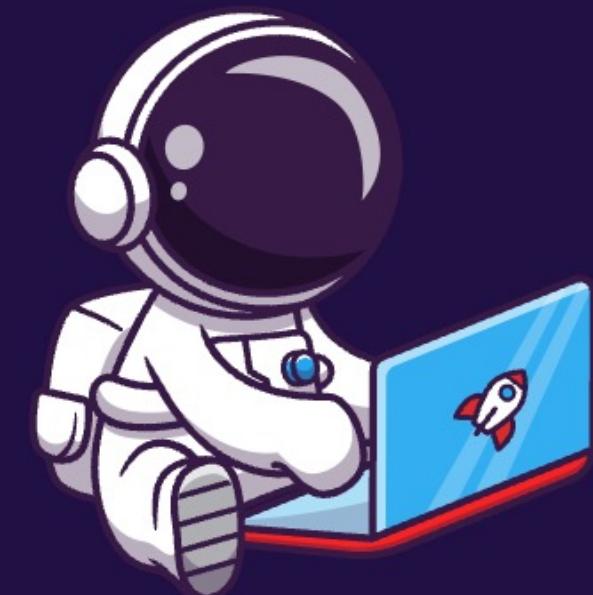
But this is safe! Redux Toolkit uses Immer to:

- Track changes to state
- Create a new immutable copy automatically
- Let you write simpler code

Feature	React Context API	RTK
State Mutation Allowed?	No (You must return a new object)	Looks like mutation, but safe!
Update Mechanism	Manual with <code>useReducer</code> or <code>useState</code>	Auto-handled by RTK + Immer
Code Complexity	More boilerplate	Cleaner and simpler syntax
Common Mistake Risk	High (accidentally mutate state)	Low (Immer tracks changes immutably)

What's Immer?

A tiny library behind the scenes in Redux Toolkit that makes mutable code behave immutably without side effects. This is one reason Redux Toolkit is better suited for large-scale state management.



Congratulations

Thank You, See you next time !!

