

Vibe Coding Is Creating Braindead Coder

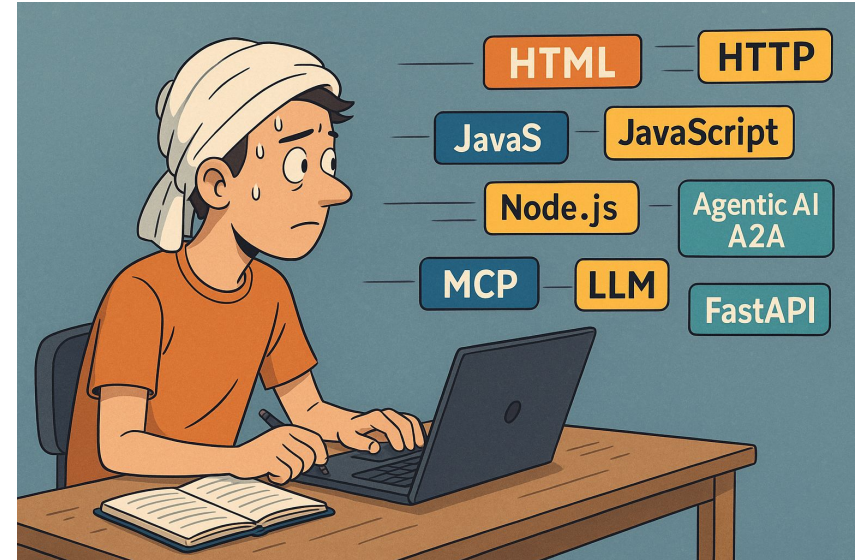
- It only works if you've been coding long enough to spot what's broken.
- It builds something that's about 80% right, 20% wrong in weird ways
 - Maybe 80% of the time it is able to fix it itself. But 20% of the time, it can get stuck chasing its own tail.
 - The last 20% of the work always takes 80% of the time
 - Then spend another hour manually fixing the last mile and testing everything yourself.
- it makes developers crave instant gratification instead of deep understanding, and reduces us to gamblers who pull levers for the next hit of working code.
- The solutions from AI coding agents are more complex than if a human had coded them. The code they generate often is **duplicative**. It frequently handles corner cases that are not at all likely.

Vibe Coding is gonna spawn the most braindead software generation ever

But three weeks later, the dashboard starts to choke whenever more than 50 users go online. Sarah's got no clue why because the AI barfed out thousands of lines of code she has never eyeballed and couldn't decode even if her life depended on it. She crawls back to Lovable prompt, begging "*fix the crashy thingy please*" but the AI's just as clueless as she is about the bug that is emerging from some **twisted tango** between database queries, memory management, and concurrent user sessions.

Do you like coding?

- Hws: Manual coding
- Lab 1 and Lab 2: Manual Coding
- Group Project: Manual + AI Coding
- make sure to **code every day**



Modern web application development

Web evolution

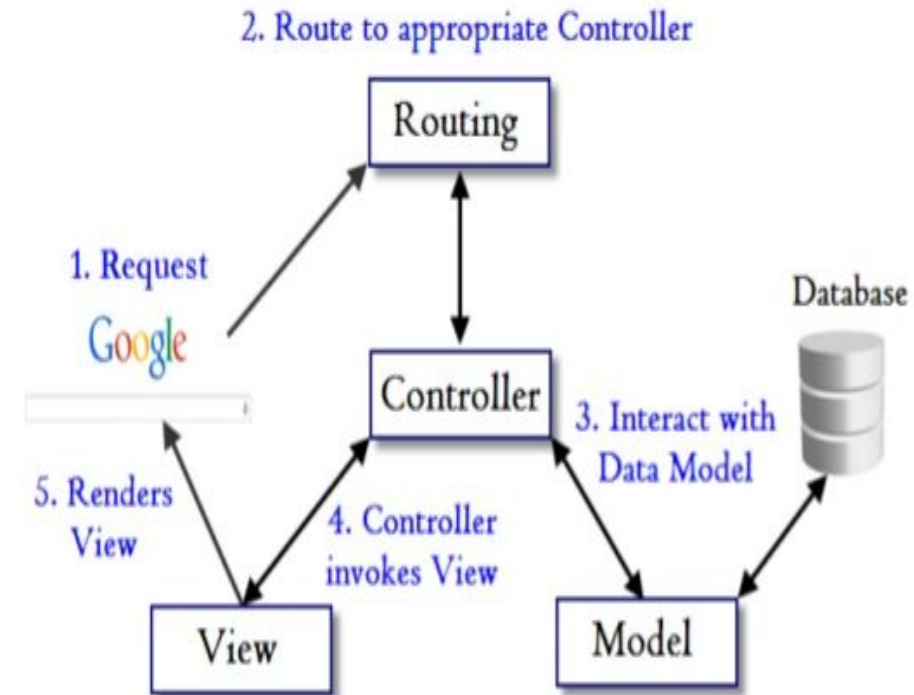
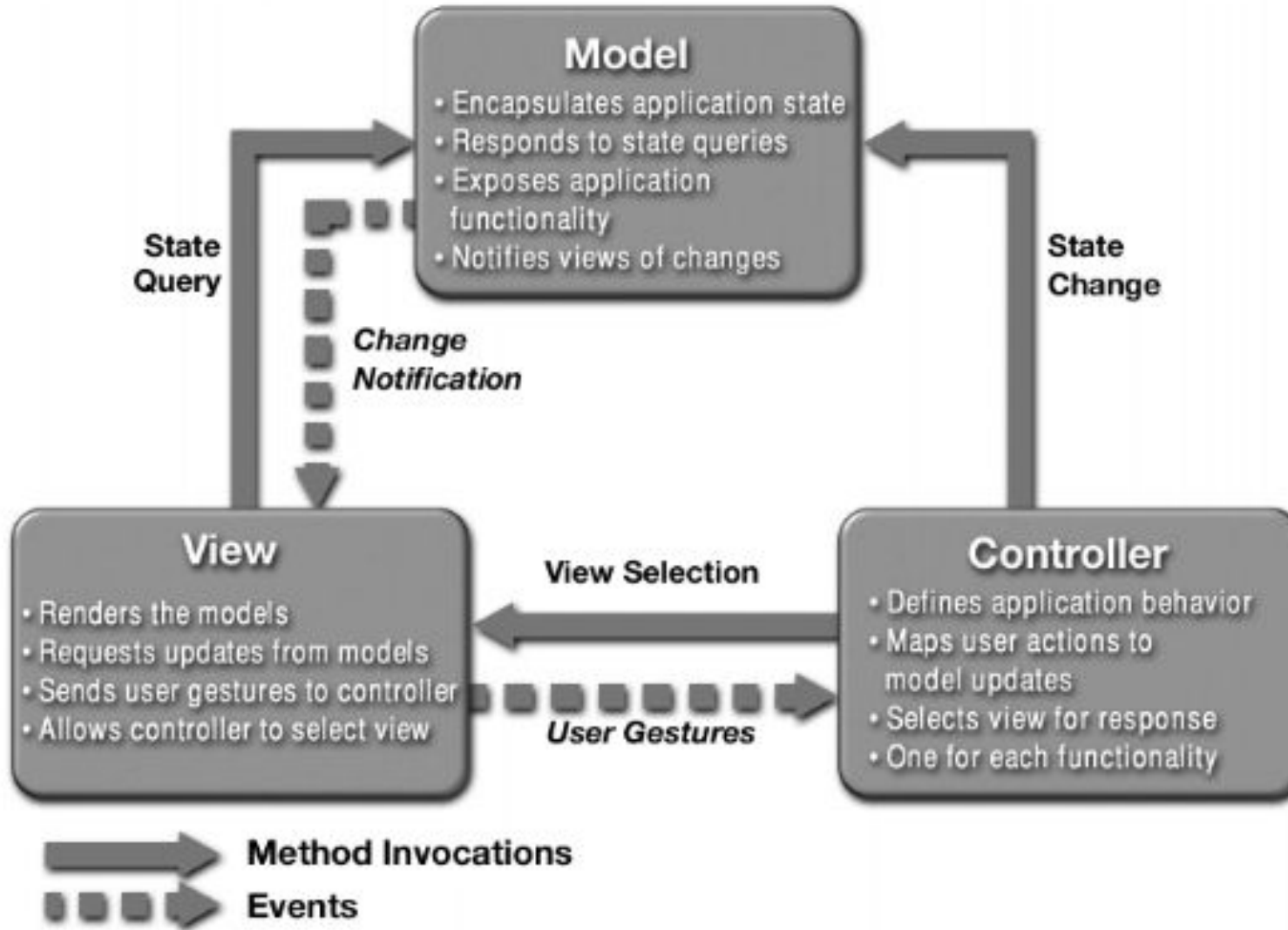
- Web has evolved from simple static pages to **complex real time** pages



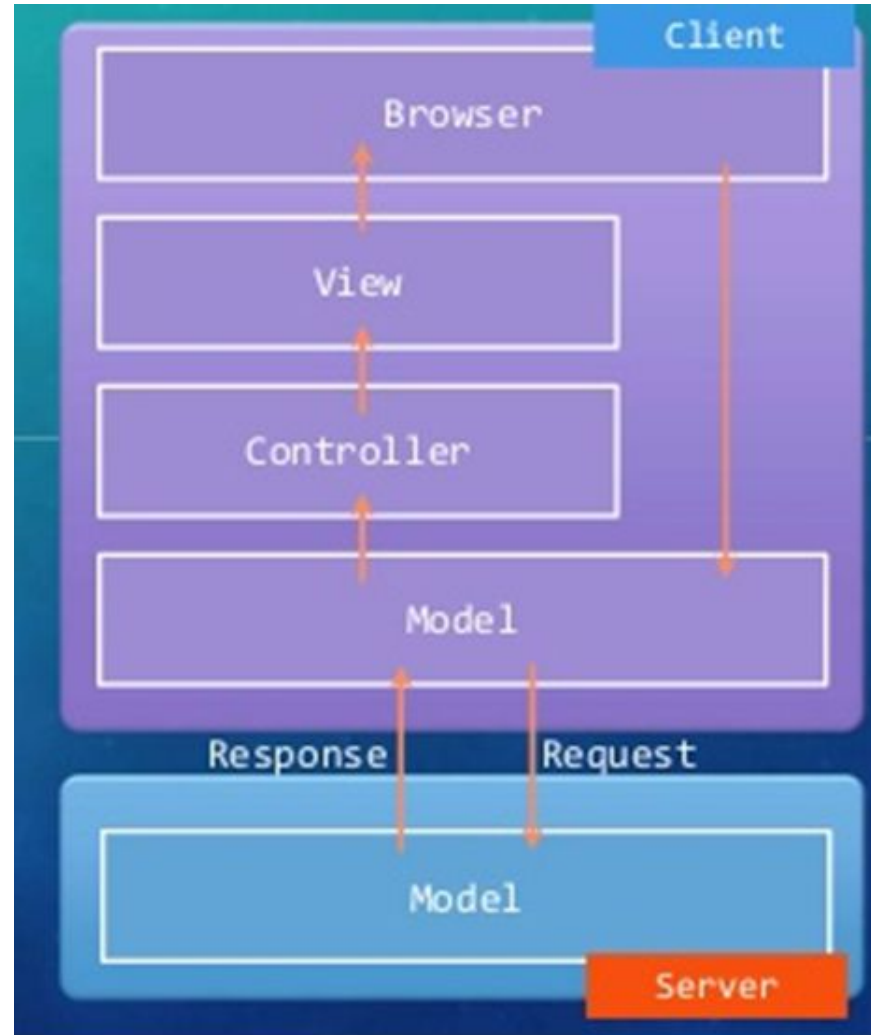
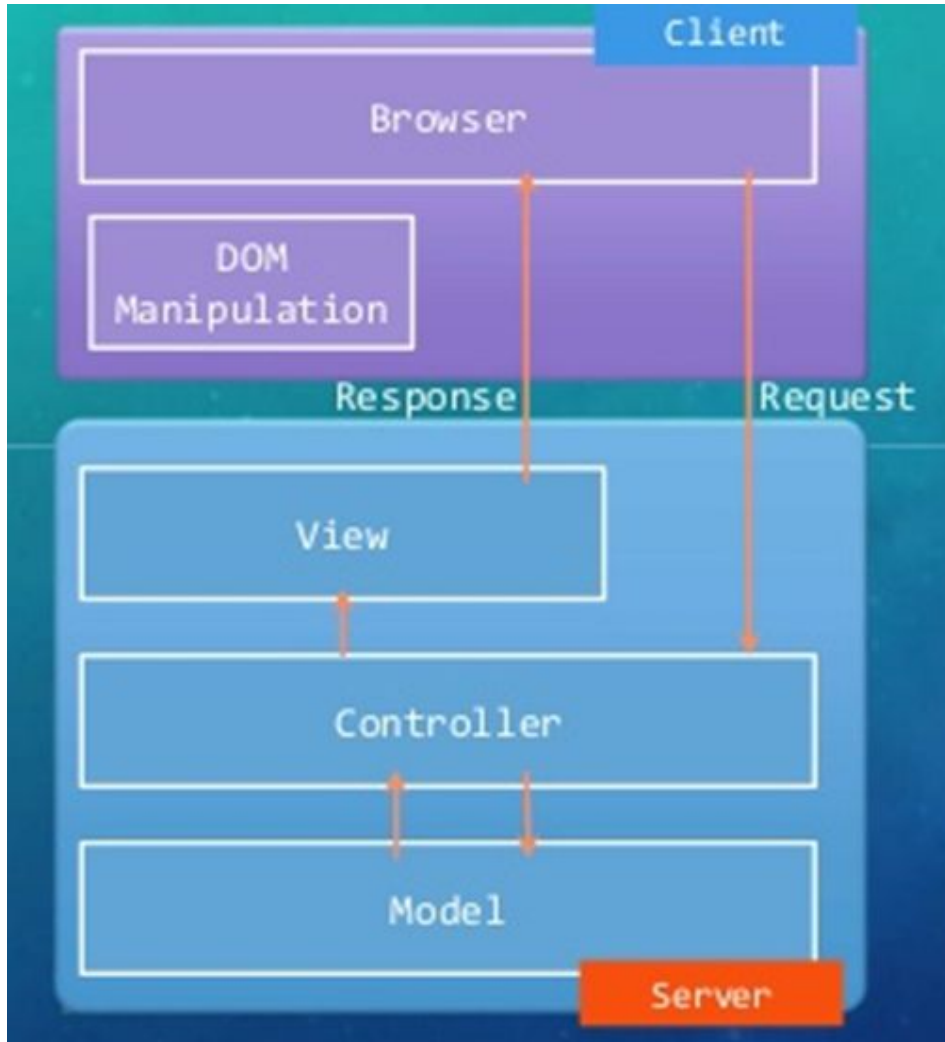
- [Arts and Humanities](#) - [Architecture](#), [Photography](#), [Literature](#)...
- [Business and Economy \[Xtra!\]](#) - [Companies](#), [Investments](#), [Classifieds](#)...
- [Computers and Internet \[Xtra!\]](#) - [Internet](#), [WWW](#), [Software](#), [Multimedia](#)...
- [Education](#) - [Universities](#), [K-12](#), [College Entrance](#)...
- [Entertainment \[Xtra!\]](#) - [Cool Links](#), [Movies](#), [Music](#), [Humor](#)...
- [Government](#) - [96 Elections](#), [Politics \[Xtra!\]](#), [Agencies](#), [Law](#), [Military](#)...
- [Health \[Xtra!\]](#) - [Medicine](#), [Drugs](#), [Diseases](#), [Fitness](#)...
- [News and Media \[Xtra!\]](#) - [Current Events](#), [Magazines](#), [TV](#), [Newspapers](#)...
- [Recreation and Sports \[Xtra!\]](#) - [Sports](#), [Games](#), [Travel](#), [Autos](#), [Outdoors](#)...
- [Reference](#) - [Libraries](#), [Dictionaries](#), [Phone Numbers](#)...



MVC

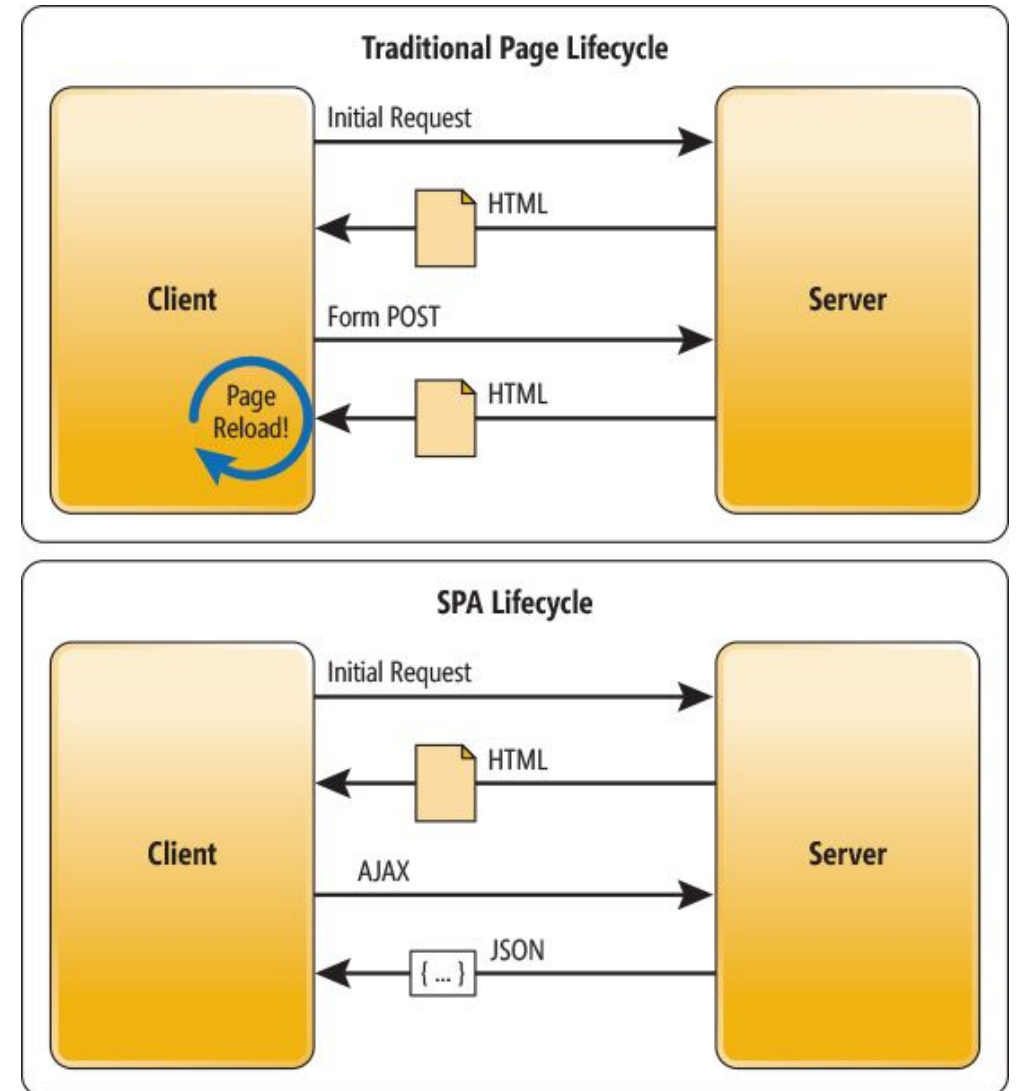


Server side MVC and Client side MVC



Single Page Application (SPA) vs MPA

- SPA interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server
- appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions
- SPA needs a Client-side routing that allows you to navigate around a web page
- SPA offers native application like experience



Single Page Application

- Advantage

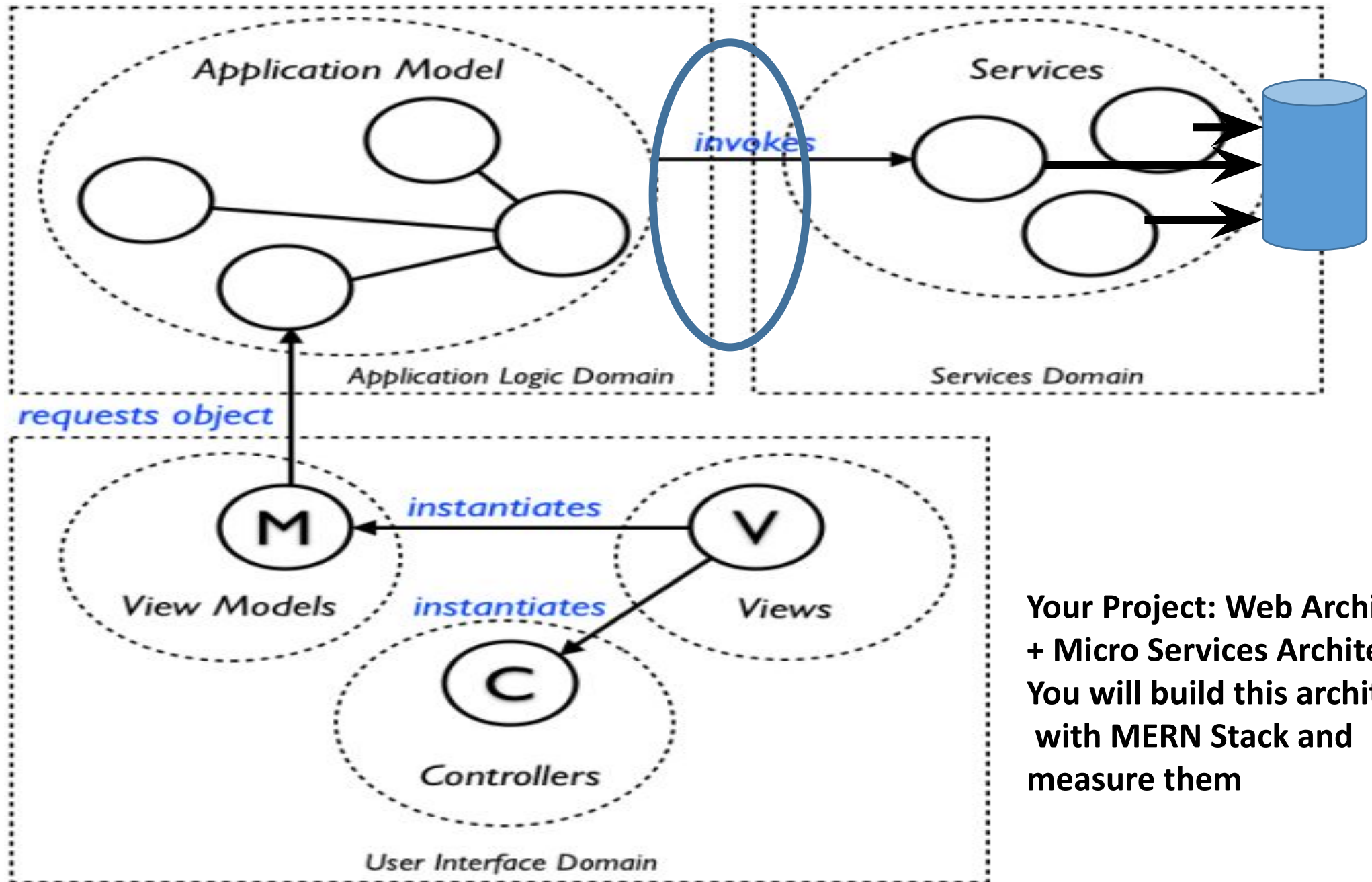
- Single Page Apps are smooth and fast.
- They are easier to develop, deploy and debug.
- Can be transited to mobile apps by reusing the same backend code.

- Disadvantages

- SPAs perform poor on the search engine. But now, with isomorphic rendering/server-side rendering even SPAs can be optimized for search engine too.
- They are less secure compared to traditional multi-page apps because of its cross-site scripting.

SPA and Client side framework

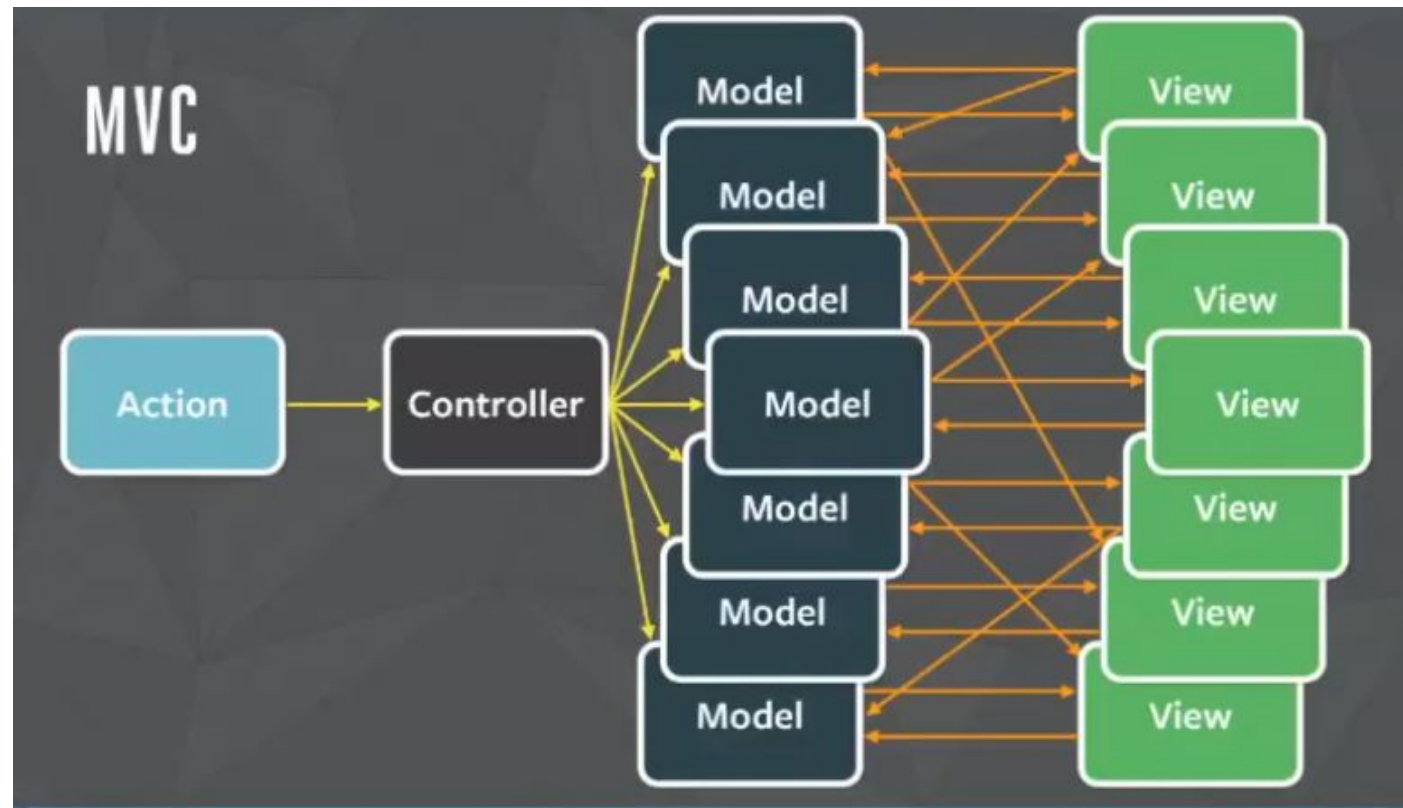
- Framework can manage large, complex application
- Client-side frameworks
- React, AngularJS, Backbone, Ember, ExtJS, Knockout, MetroJS, Vue.JS, etc.
- React
 - UI building library developed by Facebook
 - Using Virtual Dom for Dom manipulation
 - Create the new concepts such as Flux
- AngularJS
 - All-in-one web application framework developed by Google
 - Data binding, Directive, Routing, Security etc



**Your Project: Web Architecture
+ Micro Services Architecture
You will build this architecture
with MERN Stack and
measure them**

Problems with old technology

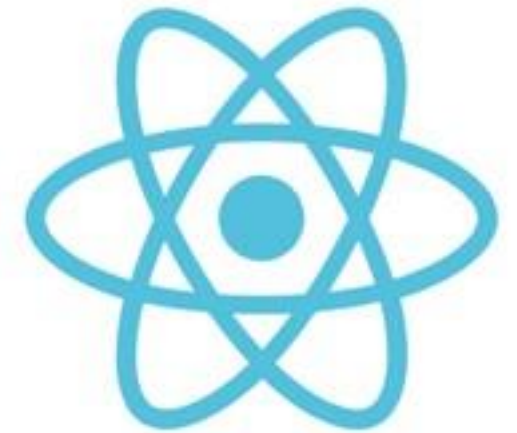
- MVC does not scale
- Typical MVC architecture



Problems with old technology (Contd)

- Complexity of two-way data binding
- Bad UX from using "cascading updates" of DOM tree
- A lot of data on a page changing over time
- Complexity of Facebook's UI architecture

Client-side frameworks



- ReactJS <https://facebook.github.io/react/>
- <https://reactjs.org/>
- **Javascript library for building user interfaces**
- **Components** (collection of HTML, CSS, JS) **are the building blocks of React**
- React is a JavaScript framework that focuses on declarative syntax and virtualization of DOM.
- It is an open-source Javascript library Developed by Meta in 2013.
- It provides a declarative and efficient way to create interactive UI components.
- It allows building more reusable and maintainable UI components with ease.

.

Key concepts

- **JSX:** A syntax extension that allows writing HTML-like code within JavaScript. It makes UI development more readable and easier to manage in React.
- **Components:** The building blocks of a React application. Components are reusable, independent, and can be either functional or class-based.
- **Props:** Read-only data passed from parent components to child components, allowing for dynamic and customizable content
- **State:** An internal data store within a component. When the state changes, the component re-renders to reflect the updates.
- **Unidirectional data flow:** Data in React flows in a single direction, from parent to child components, ensuring predictable and maintainable state management.
- **Virtual DOM:** A lightweight copy of the actual DOM that React uses to efficiently update the UI. It minimizes direct manipulations of the real DOM, improving performance.
- **Hooks:** Built-in functions that allow functional components to use state and other React features without needing class components. Examples include `useState` and `useEffect`.
- **React Router:** A library that enables navigation and routing in React applications, allowing users to move between different views or pages without full-page reloads.

React's popularity stems from its efficiency, reusability, and the large ecosystem of tools and libraries built around it. It is widely used in web development for creating interactive and dynamic user experiences.

Most Forked Repos (Click to View Repo Link on GitHub)

2015

tensorflow/tensorflow	Open source software library for numerical computation using data flow graphs.	4,355	1
facebook/react-native	A framework for building native apps with React.	4,198	2
NARKOZ/hacker-scripts	Based on a true story	3,553	3
apple/swift	The Swift Programming Language	3,068	4

Who uses React?

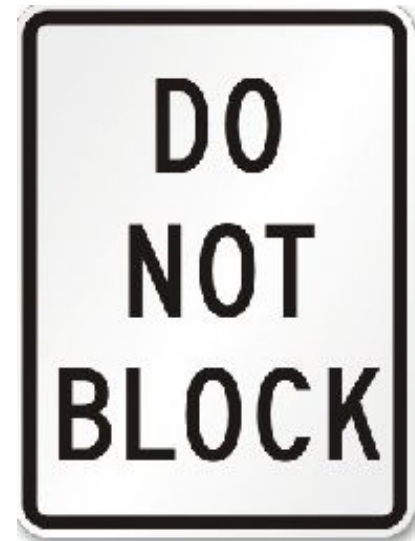
<https://jaydevs.com/top-companies-using-react-js/>

There are too many companies using React JS to list them all here, so instead we're going to cover the top 30 companies that use React JS. We'll then detail what React components they use and the benefit it has for their web and mobile application development.



Why use React?

- Easy to read and understand views
- Concept of components is the *future* of web development
- If your page uses a lot of **fast-updating data** or **real-time data**, React is the way to go



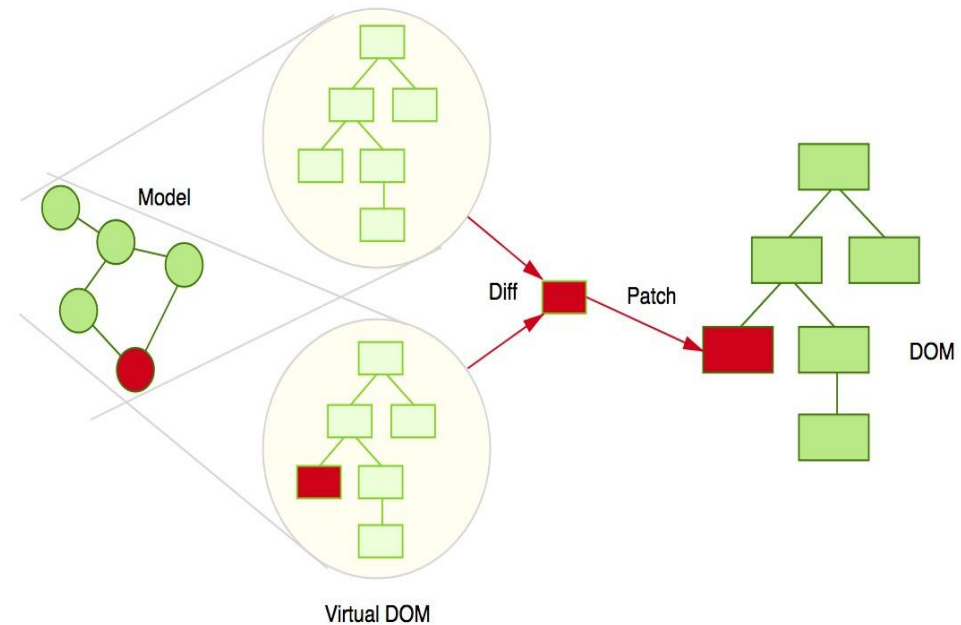
Data Mutation is problem

- When data changes, refresh
- When data changes, React re-renders the component
- But Stateful Browser DOM
- Reacts builds new virtual DOM subtrees
- Computes the minimal set of DOM mutations and puts them in a queue and Batch executes all updates

DATA CHANGING OVER TIME IS
THE ROOT OF ALL EVIL

ReactJS

- Virtual DOM. Keeping state of DOM is hard
- Efficient diff algorithm
- Batched update operations
- Efficient update of sub tree only
- Uses observable instead of dirty checking to detect change
- AngularJS uses dirty checking that runs in a cycle after a specified time, Checking the whole model reduces the performance and thus makes the application slow.




React: 50% better performance

MOBILE SITE SEARCH

React rewrite

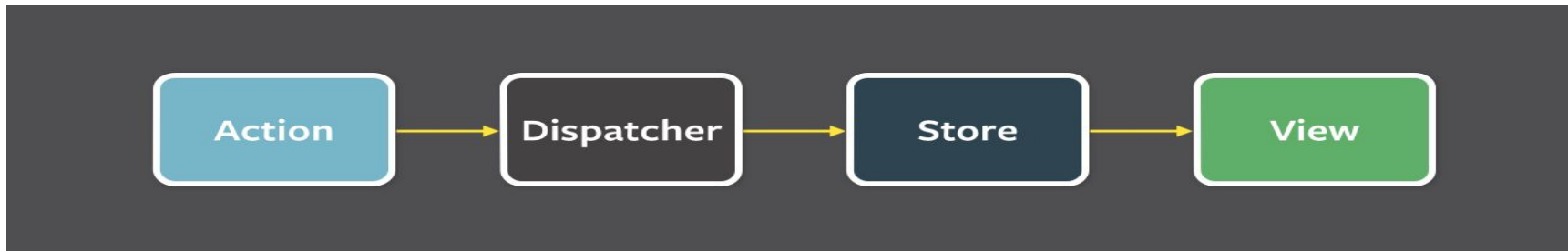
- More features, same amount of time
- With 50% better performance



Source: Facebook

Two-way vs. Unidirectional data flow

- In two-way data binding
 - The view is updated when the state changes, and vice versa.
 - For example, when you change a model in AngularJS, the view automatically reflects the changes.
 - It can lead to cascading updates and changing one model may trigger more updates.
 - View ↔ State
- Unidirectional data flow
 - Mutation of data is done via actions. So, new data always enters into the store through actions.
 - View components subscribe to the stores and automatically re-render themselves using the new data. So, the data flow looks like this :

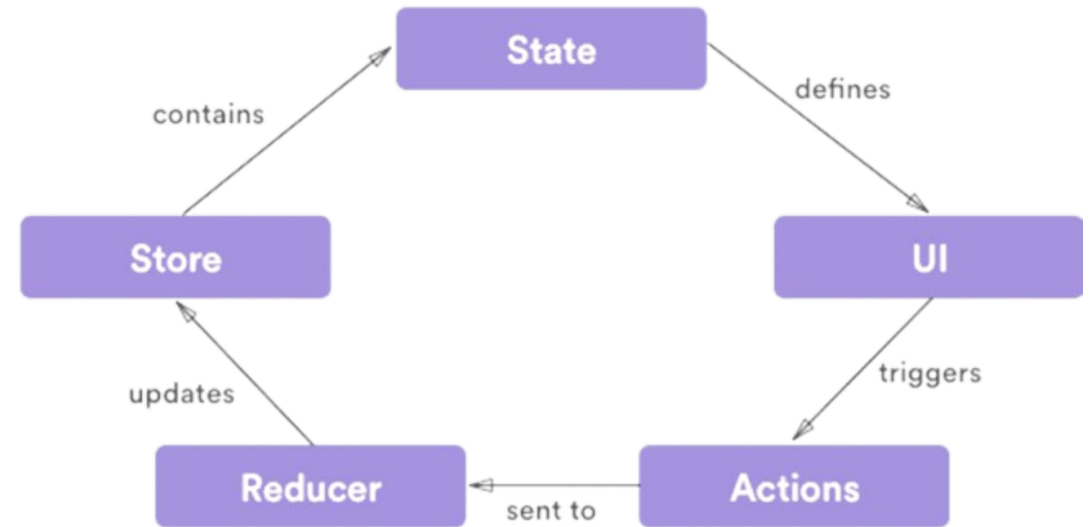


Source: Facebook

React

- **State Management**

- State management is a critical aspect of building complex web applications.
- ReactJS provides several options for managing state Efficiently within its ecosystem.
- Various ways of State Management:
 - Local Component State
 - `useState`
 - Context API
 - Redux



What is a React component?

- ReactJS follows a component-based architecture, where UI are composed of reusable building blocks called components.
- A React component is a JavaScript function or class that returns a JSX (JavaScript XML) representation of the UI.
- It encapsulates the UI logic and state, making it reusable and modular.
- React supports two ways of declaring a component-
- Class Components (old)
- Functional Components (new)

Functional Components (Recommended)

- A simple JavaScript function that returns JSX.
- Can use **Hooks** (`useState`, `useEffect`, etc.) to manage state and lifecycle.
- More concise and easier to understand.

♦ Example: Functional Component with Props & State

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

function Greeting({ name }) {
  const [message, setMessage] =
    useState("Welcome!");

  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>{message}</p>
      <button onClick={() => setMessage("Have a great
day!")}>Click Me</button>
    </div>
  );
}

export default Greeting;
```

Why Functional Components are Preferred Over Class Components in React?

1. Simplicity & Readability

- Cleaner, easier to read & write
- No need for `this` or complex class-based syntax

2. Reusability & Modularity

- Small, composable functions improve code maintainability
- Encourages better separation of concerns

3. Improved Testability

- Pure functions ensure predictable output
- Easier unit testing with consistent results

4. Better Performance

- No class overhead, reducing memory usage
- Optimized by React's rendering process

5. Hooks

- `useState`, `useEffect`, and custom hooks bring powerful features
- Enables state management in functional components

6. Conciseness & Maintainability

- Less code, fewer bugs
- Simplifies state & lifecycle management

Class Components Still Exist: Useful in legacy codebases, but modern React favors functional components with hooks for cleaner, more efficient development.

Structure of React Component

- Props (Properties)
 - Props allow passing data from parent components to child components.
 - Props are read-only and should not be modified within the component.
 - Props are read-only (immutable) and cannot be modified within the component.
 - Props help in passing dynamic data and event handlers.
 - Props are accessed inside components via props.name.
 -

Example: Passing Props to a Component

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

```
function App() {  
  return <Welcome name="Alice" />;  
}
```

```
export default App;
```

Structure of React Component

- State
 - State represents mutable data that belongs to a component. Unlike props, state can change over time, triggering a re-render of the component.
 - Functional components can also have state using React hooks like `useState`.

State in Functional Components (Modern Approach using Hooks)

```
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>
  );
}

export default Counter;
```

Structure of React Component

- Styling
 - Styling can be applied to React components using CSS classes, inline styles, or CSS-in-JS libraries.
 - CSS classes can be added using the className attribute in JSX.
 - Inline styles can be applied using the style attribute in JSX.

Using CSS Classes (Recommended for maintainability)

```
import './styles.css';

function StyledComponent() {
  return <h1 className="heading">Hello, World!</h1>;
}

export default StyledComponent;
```

CSS-in-JS (Styled Components) allow writing CSS directly inside JavaScript files

```
import styled from "styled-components";

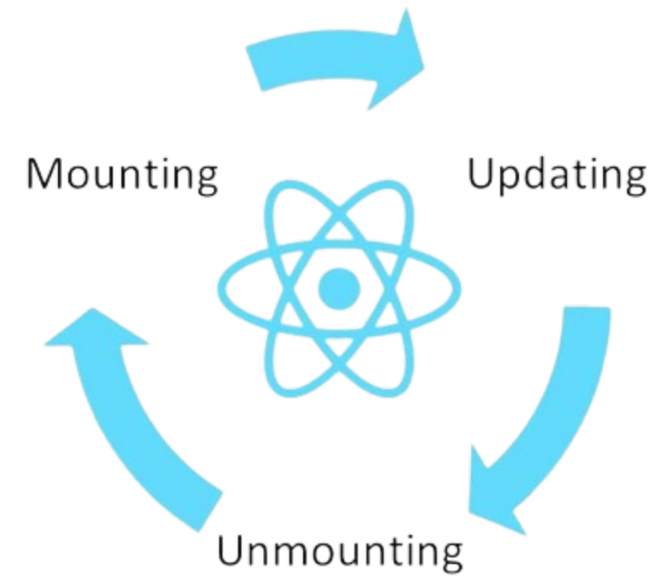
const StyledButton = styled.button`
  background-color: blue;
  color: white;
  padding: 10px;
  border-radius: 5px;
`;

function App() {
  return <StyledButton>Click Me</StyledButton>;
}

export default App;
```

Component Life Cycle

- React components have a life cycle consisting of different phases and methods that are executed at s
 - The mounting phase is when a new component is created and inserted into the DOM.
 - The updating phase is when the component updates or re-renders. This reaction is triggered when the props are updated or when the state is updated.
 - The last phase within a component's lifecycle is the unmounting phase, when the component is removed from the DOM.



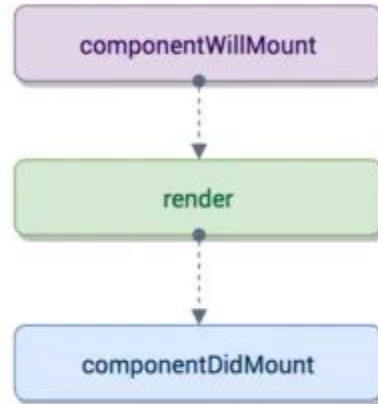
Life Cycle Hooks

- `componentDidMount()`
 - This hook is called after a component is mounted (rendered for the first time) in the DOM.
- `componentDidUpdate(prevProps, prevState)`
 - This hook is called after a component is updated and re-rendered in response to changes in props or state.
- `componentWillUnmount()`
 - This hook is called just before a component is unmounted and removed from the DOM.
- `shouldComponentUpdate(nextProps, nextState)`
 - This hook is called before a component is re-rendered and allows you to control if the re-rendering should occur.

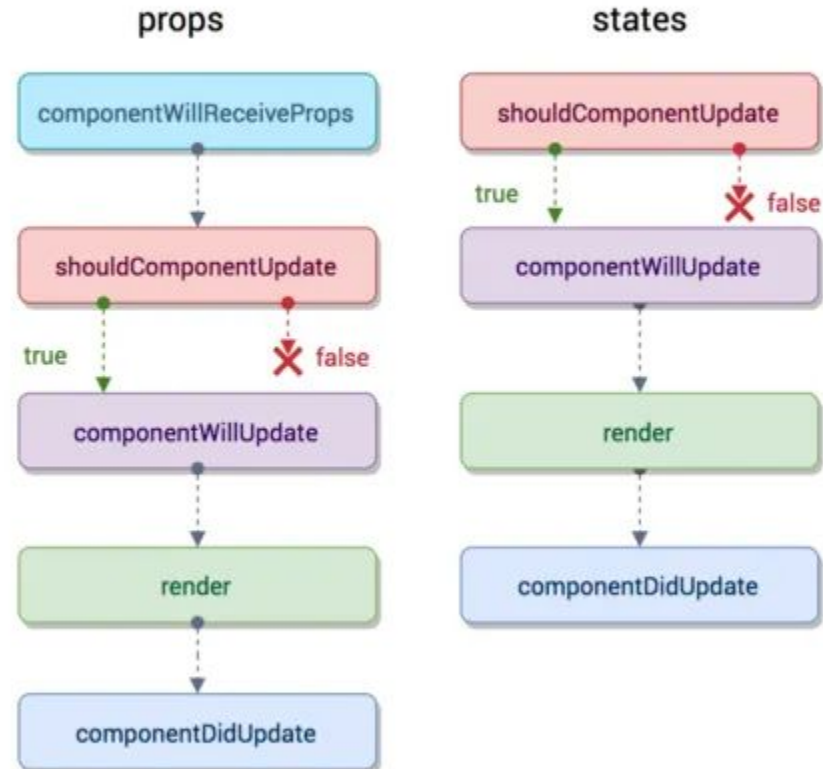
Initialization

setup props and state

Mounting



Updation

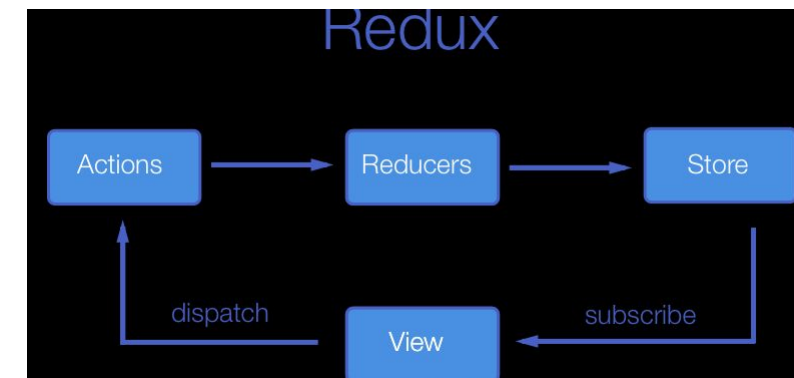
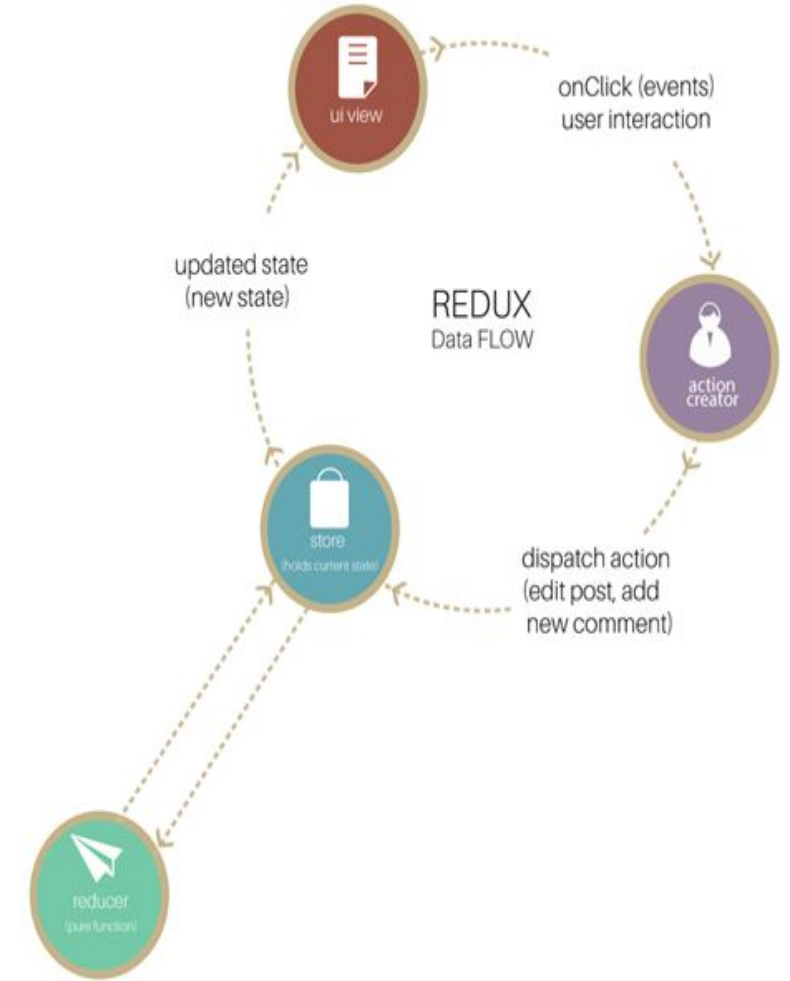


Unmounting



Redux

- Redux is a state container for JavaScript apps, often called a Redux *store*. It stores the whole state of the app in an immutable object tree.
- Actions: Actions are how the application interacts with the store. The application sends an action with some attached data. These actions are then handled by a so-called **reducer**.
- Store: This structure stores the complete state of our application. Given only the content of the store, the application should be able to save and recover the current state of the application
- Reducer: The Reducer is a function that takes the current state and an action and transforms it to a new state.



Redux

- The state is read-only: The state of the application can only be changed via actions. This is very useful for debugging since the changing state flow is much more clear.
- The Redux store API is tiny and has only four methods:
 - `store.getState()`—Returns the current state object tree.
 - `store.dispatch(action)` - Dispatch an action to change the state.
 - `store.subscribe(listener)` - Listen to changes in the state tree.
 - `store.replaceReducer(nextReducer)` - Replaces the current reducer with another. This method is used in advanced use cases such as code splitting

Simple Redux Tutorial: <https://appdividend.com/2017/08/23/redux-tutorial-example-scratch/>


JSX (JavaScript XML)

- JSX (JavaScript XML) is a syntax extension for
- JavaScript is used in React.
- JSX allows you to write HTML-like code within JavaScript, making it easier to create and manipulate the UI.
- JSX allows embedding JavaScript expressions within curly braces {}.
- JSX needs to be compiled into plain JavaScript to be understood by the browser.

```
class HelloWorld extends React.Component {  
  render() {  
    return (  
      React.createElement(  
        'h1',  
        {className: 'large'},  
        'Hello World'  
      )  
    );  
  }  
}
```

<https://reactjs.org/tutorial/tutorial.html>

<https://www.w3schools.com/REACT/DEFAULT.ASP>

 [Docs](#) [Tutorial](#) [Community](#) [Blog](#) [GitHub](#) v15.6.1

TUTORIAL

[Before We Start](#)
What We're Building
Prerequisites
How to Follow Along
Help, I'm Stuck!

[Overview](#)
What is React?
Getting Started
Passing Data Through Props
An Interactive Component
Developer Tools

[Lifting State Up](#)
Why Immutability Is Important
Functional Components
Taking Turns
Declaring a Winner

[Storing A History](#)

Tutorial: Intro To React

[Edit on GitHub](#)

Before We Start

What We're Building

Today, we're going to build an interactive tic-tac-toe game.

If you like, you can check out the final result here: [Final Result](#). Don't worry if the code doesn't make sense to you yet, or if it uses an unfamiliar syntax. We will be learning how to build this game step by step throughout this tutorial.

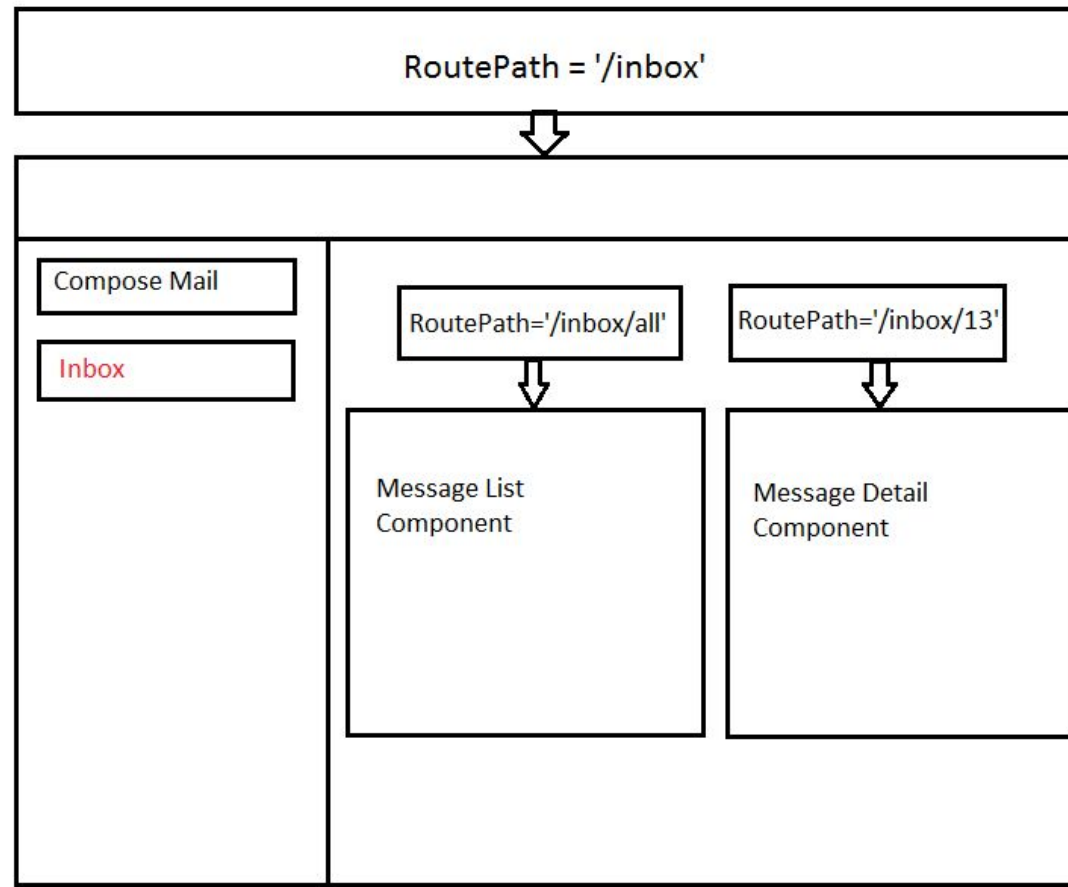
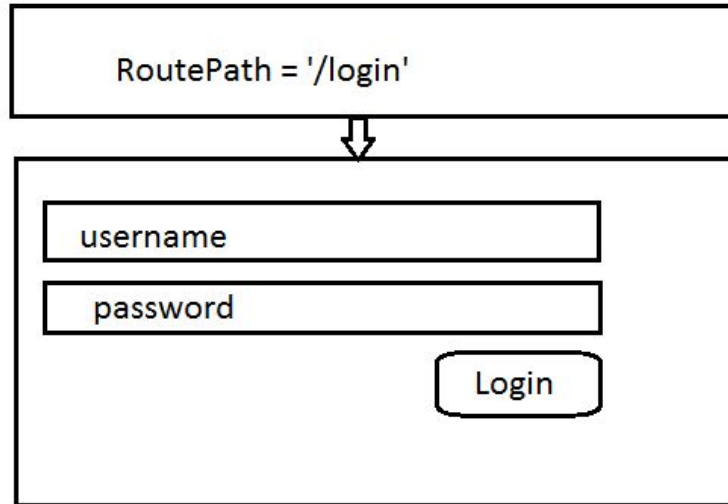
Try playing the game. You can also click on a link in the move list to go "back in time" and see what the board looked like just after that move was made.

Once you get a little familiar with the game, feel free to close that tab, as we'll start from a simpler template in the next sections.

Prerequisites

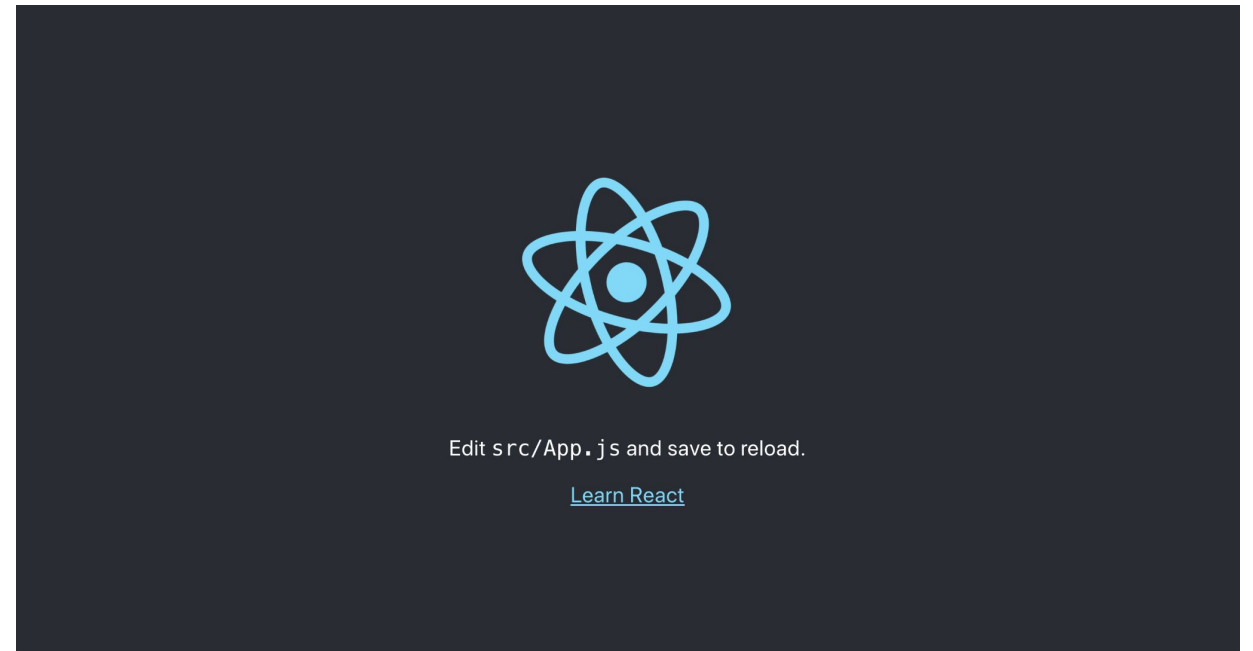
We'll assume some familiarity with HTML and JavaScript but you should be able to follow along even if you haven't used them before.

Composition of Components



Running a React Application

- To run a React Application, we need to install
 - NodeJS
 - Node Package Manager (NPM)
- Install create-react-app
 - `npm install -g create-react-app`
- Create React Application
 - `npx create-react-app my-app`
- Run development server
 - `cd my-app`
 - `npm start`
- Build Project
 - `npm run build`



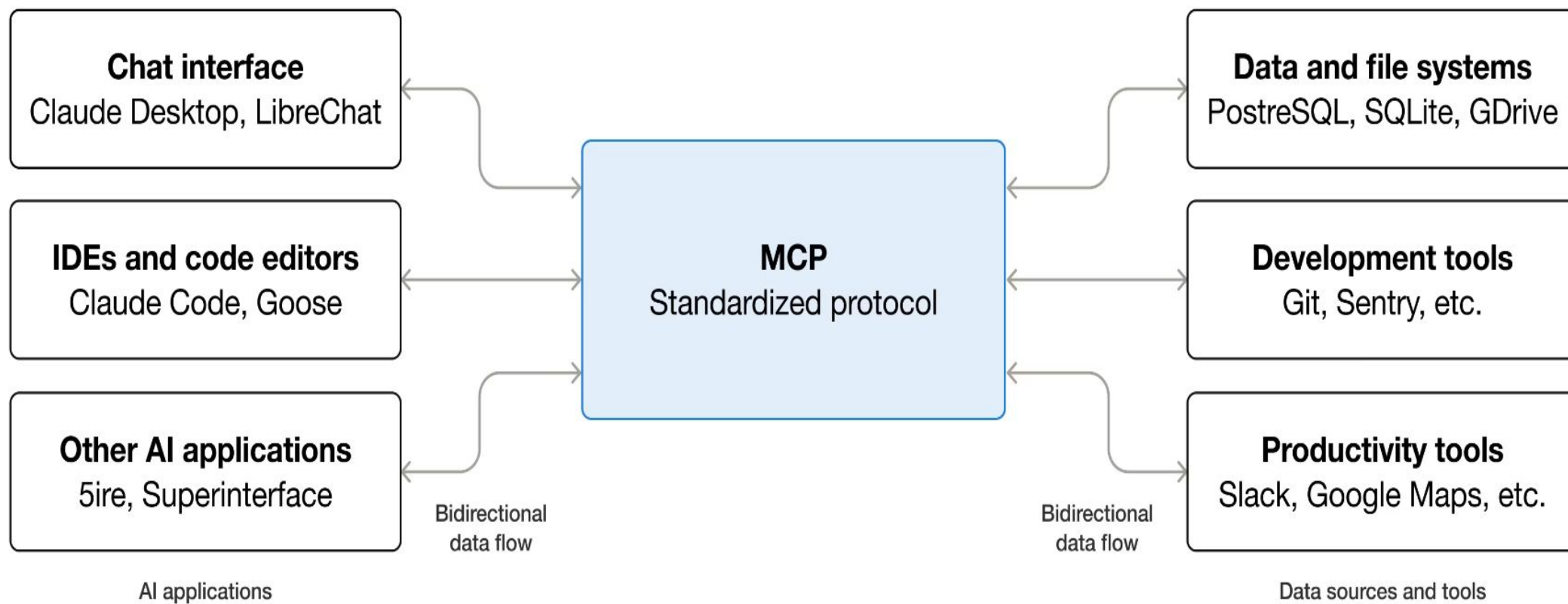
Basics of the Model Context Protocol (MCP)

Why MCP matters to you

- One integration, many models & apps (portable skills) Build assistants that can **act** (not just chat) Standardized primitives → easier team projects
- In-demand skill across IDEs & AI tooling Perfect for portfolio demos & hackathons

What is MCP?

- Introduced by [Anthropic](#)
- An **open protocol** that lets LLM apps discover & use:
 - **Resources** (context/data to read, think “GET-like” endpoints)
 - **Tools** (actions/functions to call, think “POST-like”)
- **Prompts** (reusable templates/workflows)
- ...and communicate over a simple client–server interface.
- LLM clients (e.g., chat apps, IDE assistants) connect to one or more MCP servers



Architecture

- Host app (e.g., Claude Desktop, IDE)
- MCP client (built into the host, discovers/uses servers)
- MCP servers (expose Resources (read data), Tools (do actions), Prompts (reusable templates)).
- Transports:stdio (local) or Streamable HTTP (remote)
- Messages: all requests/responses are JSON-RPC 2.0.

Transports you'll meet

- **stdio**: launch server as a subprocess; JSON-RPC via stdin/stdout
- **Streamable HTTP**: one endpoint for POST/GET; can stream with SSE Custom transports are possible if you keep JSON-RPC & lifecycle rules
- JSON-RPC is a very lightweight, text-based protocol for making remote procedure calls using JSON as the data format.

```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "method": "subtract",  
  "params": { "minuend": 42, "subtrahend": 23 }  
}
```

MCP Transport: Streamable HTTP (What & How)

- **Single MCP endpoint** (e.g., `/mcp`) supports **POST** (send) and **GET** (listen).
- Client **sends** JSON-RPC via **HTTP POST** and includes
- Accept: `application/json, text/event-stream`
- POST body can be:
 - a single JSON-RPC request/notification/response, or
 - a **batch** (array) of requests/notifications/responses.
- If POST has **only** responses/notifications → server returns **202 Accepted**.
- If POST includes **requests** → server responds with **either**:
 - Content-Type: `application/json` (single JSON object), or
 - Content-Type: `text/event-stream` (SSE (Server-Sent Events) stream) that can include the responses (and related server messages)

Streamable HTTP: Listening, Reliability & Security

- Client may **open a GET** to the MCP endpoint with
- *Accept: text/event-stream* to listen for server messages; server returns SSE or **405** if not supported.
- On SSE (Server-Sent Events) streams:
 - Server may send **JSON-RPC requests/notifications** (e.g., server-initiated prompts); **must not** send responses here except when resuming.
 - **Multiple streams** are allowed; server must not broadcast the same message to multiple streams.
- **Session management:** server may set *Mcp-Session-Id* at init client echoes it on all requests; *DELETE* can end a session; 404 on a session → client re-init

The 3 core primitives

1) Resources (read context)

Think: “GET-like” data the model can load into context

Examples: files, docs, database rows, URLs

2) Tools (do things)

Think: “POST-like” actions with inputs/outputs Examples:

create issue, run query, write file

3) Prompts (reusable flows)

Parameterized templates exposed by servers (often user-initiated)

A simple flow (from the model's POV)

1. User asks a question
2. Client discovers needed **resource/tool/prompt**
3. UI asks for permission (human-in-the-loop)
4. Client calls the server (MCP message)
5. Server returns data/results
6. Model answers using the fresh context

example idea

“Summarize files in my project folder.”

- Run a **filesystem MCP server**
- Client requests permission to read a folder
- Server streams back file contents/links as **resources**
- Model summarizes with citations

Capability discovery (initialisation)

- On startup, client asks server for:
 - **tools/list**, **resources/list**, **prompts/list** (and updates)
 - Client presents these in UI (slash commands, pickers, etc.)
- Model can autonomously choose tools; prompts are typically user-triggered
 - these are predefined templates or workflows (like “Draft email,” “Summarize file,” “Generate PR description”)

Useful client features (you'll encounter)

- **Roots**: limit filesystem access to safe directories

```
"roots": [  
  { "uri": "file:///Users/alex/projects/demo", "name": "Demo Project" }  
]
```

- **Sampling**: servers can request an LLM completion from the client *"method": "sampling/createMessageCompletion",*

- **Elicitation**: servers ask the user for extra input mid-workflow

```
"method": "elicitation/request",  
"params": {  
  "message": "Which database should I connect to?",  
  "choices": ["Postgres", "MySQL", "SQLite"]  
}
```

Where you actually use MCP (today)

- **Claude Desktop**: connect local/remote servers (filesystem, Git, etc.)
- **AI IDEs** (Cursor, Windsurf, Continue): one-click servers for code tasks
- **Design/dev tools** (e.g., Figma MCP server) to fetch precise design data
- <https://www.figma.com/blog/introducing-figmas-dev-mode-mcp-server/>

The Data Layer (why MCP > ad-hoc APIs)

- **Resources**: typed, addressable URIs; clients can preview, chunk, stream.
- **Tools**: invocations are **JSON-Schema** validated; safer, predictable.
- **Prompts**: server-bundled prompt templates → repeatable workflows.
- **Resource links** in results let tools hand back “attachable” context.

Transports: stdio vs Streamable HTTP

Transport	Best for	Notes
stdio	local dev, CLI tools	super simple and fast; single client
Streamable HTTP	remote servers	a single <code>/mcp</code> (or <code>/message</code>) endpoint; optional session IDs; supports streaming & reconnection

Authentication & Authorization (industry patterns)

- **HTTP transport**: bearer/API keys; **OAuth recommended** for token issuance.
- **Host controls**: consent prompts per **client tool** pair; scoping via **roots** (allowed directories).
- **Platform hardening** (e.g., Windows): proxy-mediated comms, signed servers, registries, isolation.

Why companies choose MCP

- **Design-to-code**: Figma's MCP server surfaces semantic design data → higher- fidelity codegen with Copilot/Cursor/Claude Code.
- **IDE extensibility**: Cursor/Windsurf/Continue add servers with a few clicks, unlocking web search, DBs, Sourcegraph, etc.
- **OS-level governance**: Windows pushes registry/consent/isolation to reduce tool poisoning & credential risks.
- **Enterprise roll-outs**: Copilot Studio GA integrates MCP with tracing, tool listing, admin UX.

When to use MCP (vs custom Integrations)

Use MCP when you need:

- Multi-client compatibility (Claude, Copilot, IDEs, desktop agents) Standard **tools/resources/prompts** model
- Remote deployment (Streamable HTTP) with auth Governance (registries, consent)

Use custom integrations when:

- You must stream proprietary binary protocols or need ultra-low-latency bespoke channels.
- You control both ends and will never share the integration.

What MCP means for AI Engineers

- **Connect once, integrate anywhere**: build a server once, use it across clients (IDEs, desktop agents, Copilot/Claude, etc.).
- **Standard primitives** → predictable UX: **resources** (context), **tools** (actions),
- **prompts** (workflows).
- **Production-grade flows**: progress notifications, cancellation, structured logging, and version/capability negotiation.

What MCP means for MLEs / Data teams

- **Reproducible evaluations**: tool inputs are JSON-schema'd; resources are URI- addressable → repeatable runs.
- **Observability**: structured **logging**, **progress**, and **cancellation** support long tasks & robust retries.
- **Governance & safety**: HTTP auth (OAuth/API keys), host-mediated consent,
- **roots** to scope file access, human-in-the-loop for tools.
- **Agentic workflows**: **sampling** and **elicitation** let servers request model output and user input through the client without leaking secrets.

Server Design Principles (from the spec)

- **Single-purpose & composable**: each server exposes focused capabilities; hosts orchestrate many servers.
- **Isolation by design**: servers don't see the whole conversation; hosts control cross-server
- **Negotiate capabilities** at init; evolve features without breaking clients.
- **Minimal surface, strong types**: small, well-typed inputs/outputs; explicit errors.

Quick Builder's Checklist

1. Pick an **SDK** (TS, Python, Java, C#, etc.).
2. Expose **one resource** + **one tool** + (optional) **one prompt**.
3. Start with `stdio` ; add **Streamable HTTP** when you need remote.
4. Add **progress**, **cancellation**, and **structured logging**.
5. Secure HTTP with **auth** and Origin checks; tighten **roots**.
6. Test in **Inspector**, then in a real client (IDE/desktop agent).