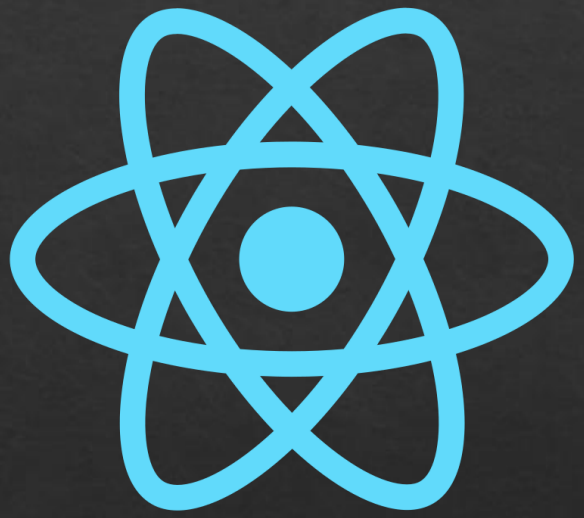
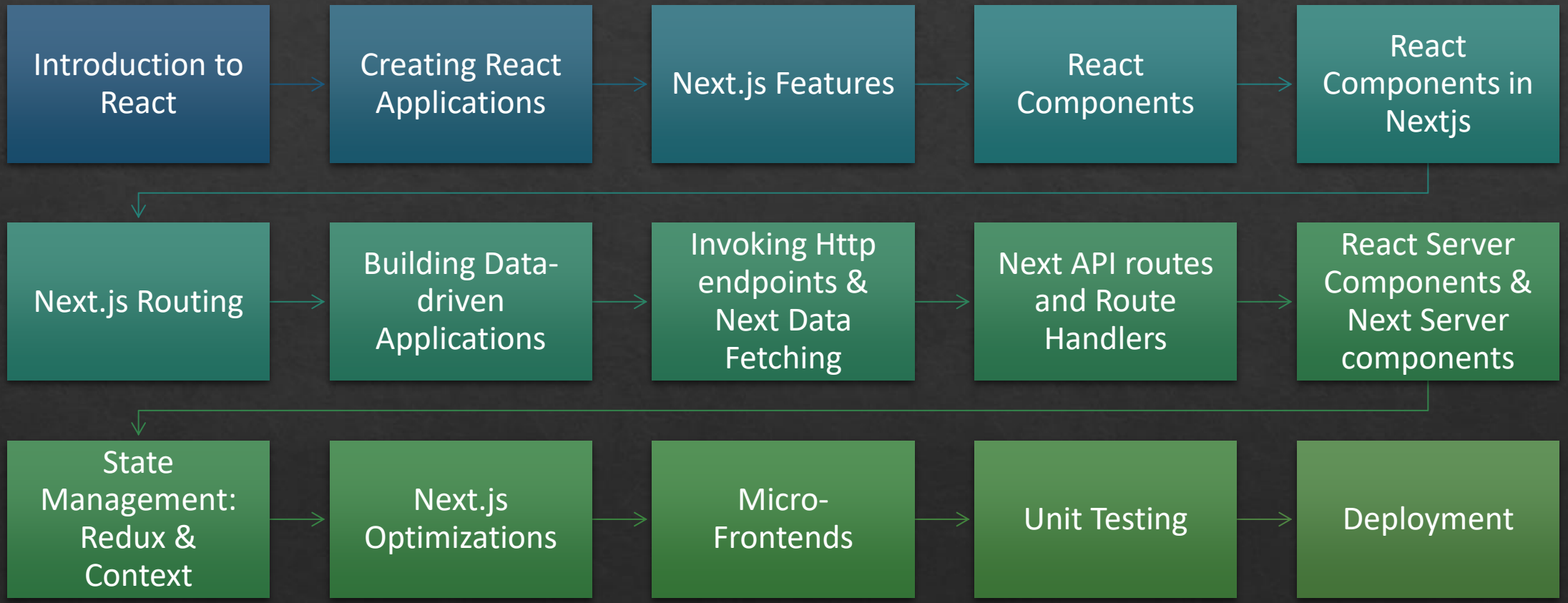
React with Next.js



ANIL JOSEPH

Presentaed by Anil Joseph(anil.jos@gmail.com)

Agenda



Presentaed by Anil Joseph(anil.jos@gmail.com)

Anil Joseph

Introduction

- Over 20 years of experience in the industry
- Technologies
  - C, C++
  - Java, Enterprise Java
  - .NET & .NET Core
  - **UI Technologies: React, Angular, jQuery, ExtJs**
  - Mobile: Native Android, React Native
- Worked on numerous projects
- Conducting trainings for corporates (700+)

Software

Node.js & NPM

HTML, CSS, JavaScript, TypeScript Editor

(Visual Studio Code)

Postman

Browsers(Chromium)

Presentaed by Anil Joseph(anil.jos@gmail.com)

JavaScript

An interpreted language

Dynamic

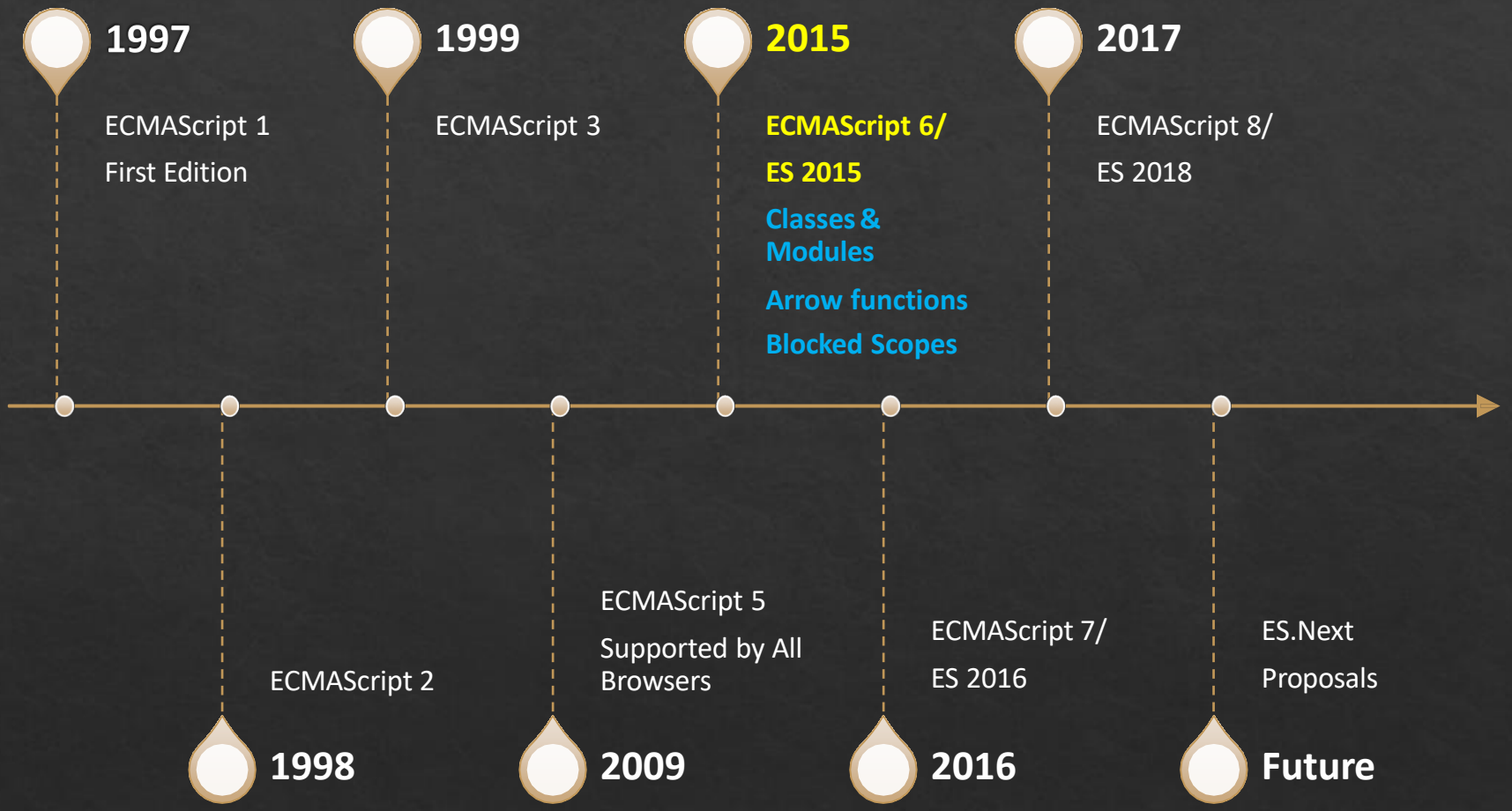
Object-oriented

Supports Functional style of programming

Available on the browsers and Node.js

Presentaed by Anil Joseph(anil.jos@gmail.com)

ECMAScript Versions

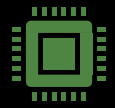


Presentaed by Anil Joseph(anil.jos@gmail.com)

Node.js

A server-side JavaScript platform

Created by Ryan Dahl in 2009



Provides an easy way to build scalable network applications

Built on top of Google V8 JavaScript Engine and libuv

NPM(Node Package Manager)

- NPM is a package manager for the JavaScript programming language.
- Allows users to consume and distribute JavaScript modules that are available on the registry
- The default package manager for Node.
- Comprises of
  - Command line client, also called npm
  - An online database called the npm registry.
    - Public
    - Paid-for private packages
  - NPM website
- The package manager and the registry are managed by npm, Inc.

NPM Packages

- Packages are reusable code published in the npm registry.
- A directory with one or more files a metadata file called package.json.
- A package is a building block that solves a specific problem.
- An application generally depends on many packages.
- Packages can be used on
  - Server side. Example: Express, Request
  - Client Side. Example Angular, React
  - Command based. Typescript, Angular CLI, JSLint

NPM Commands

- NPM is installed with Node
- Check the version
  - npm –v
- Update npm
  - npm install npm@latest –g
- Find the path to npm's directory:
  - npm config get prefix
- Help
  - npm –h
  - npm [command] –h
  - npm help [command]
  - npm help-search [key]

package.json

- A metadata file for the project
- Used to track dependencies
- Create Scripts
- Command to create package.json
  - npm init
- Setting defaults
  - npm set init-author-name ‘Anil Joseph’
  - npm get init-author-name
  - npm config delete init-author-name

Installing Packages

- Install to a project
  - npm install express
- Install and save to dependencies
  - npm install angular --save
- Install and save to development dependencies
  - npm install express --save –dev
- Install globally
  - npm install gulp -g

Install Packages with version

- Specific version
  - npm install underscore@1.8.2
- Latest Version
  - npm install underscore@1.8.x
  - npm install underscore@1.8
  - npm install underscore@1.8.2
  - npm install underscore@1.x
  - npm install underscore@1
  - npm install underscore
- Other Options
  - npm install underscore@”>=1.4.x < 1.6.x”

TypeScript

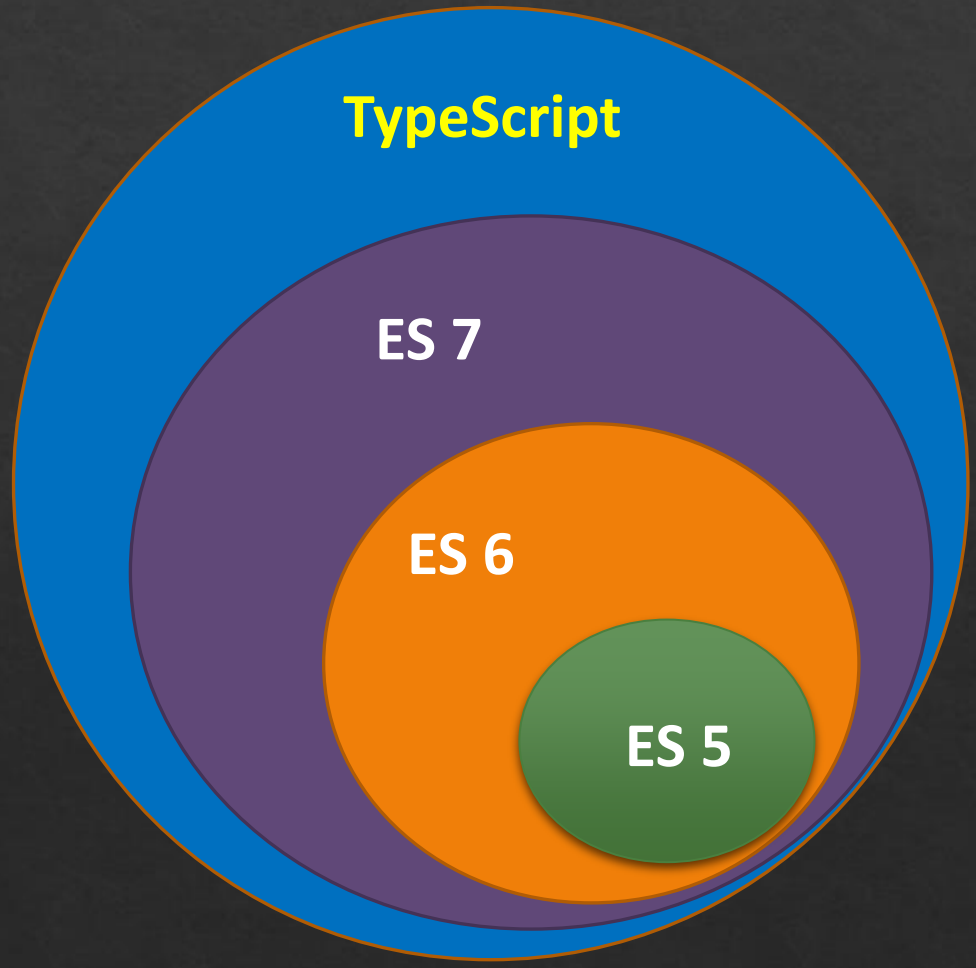
TypeScript is programming language developed and maintained by Microsoft.

TypeScript is a typed superset of JavaScript.

Transcompiles to JavaScript.

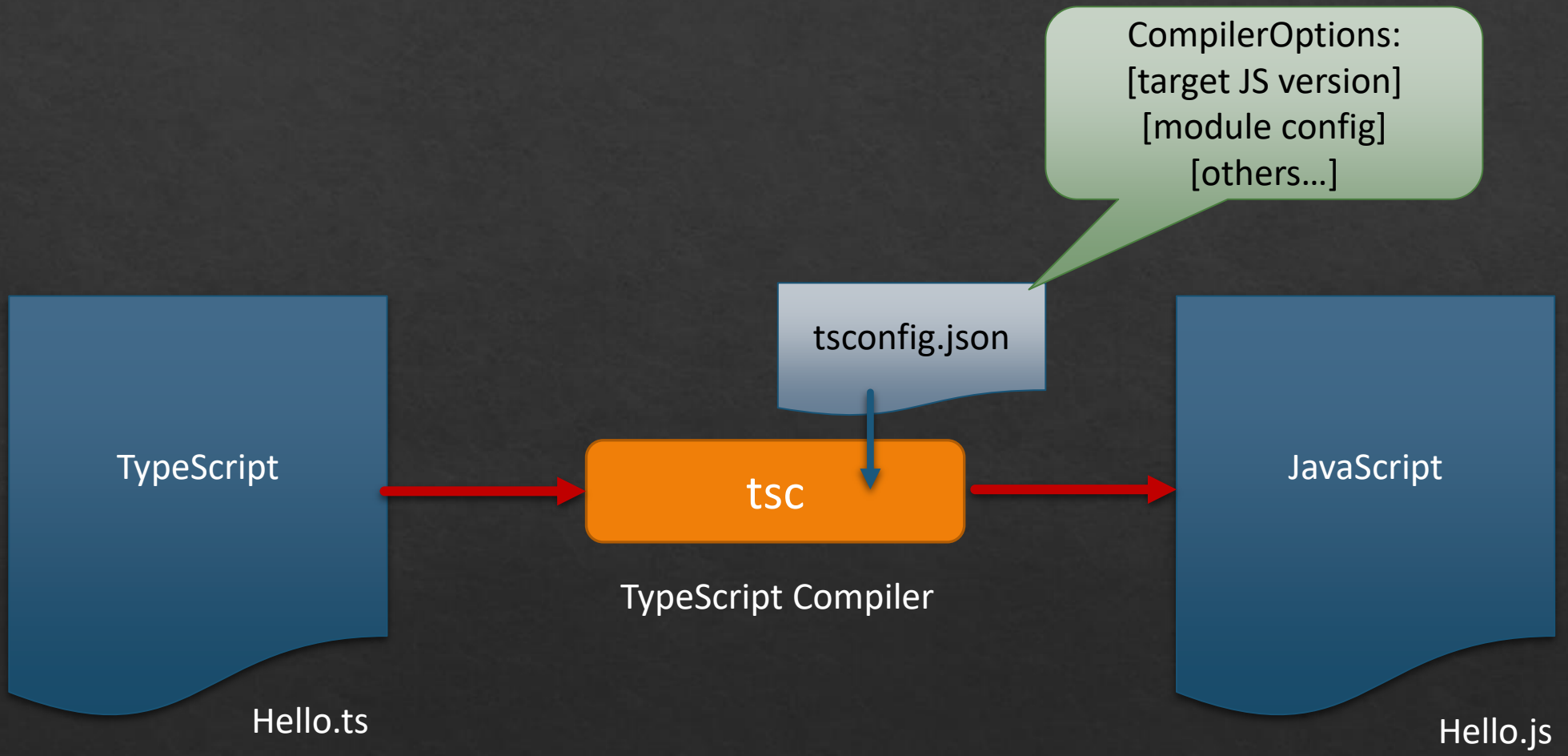
Designed for development of large applications.

Open Source.



Presentaed by Anil Joseph(anil.jos@gmail.com)

TypeScript



Presentaed by Anil Joseph(anil.jos@gmail.com)

TypeScript Features

- Type Annotations
- Compile-Time Type Checking
- Type Inference
- Interfaces
- Classes and Inheritance
- Namespaces and Modules
- Generics
- Decorators
- Arrow Functions

Presentaed by Anil Joseph(anil.jos@gmail.com)

TypeScript Installation

- Install NodeJs and NPM
- Run the command **npm install -g typescript**

Presentaed by Anil Joseph(anil.jos@gmail.com)

TypeScript Types

Boolean

- **let** isAvailable:boolean = false

Number

- **let** age: number = 16;

String

- **let** name: string = “Anil";

Array

- **let** list: number[] = [1, 2, 3];
- **let** list: Array<number> = [1, 2, 3];

Presentaed by Anil Joseph(anil.jos@gmail.com)

TypeScript Types

Enum

- **enum** Color {Red, Green, Blue}
- **let** c: Color = Color.Green;

Any

- **let** x: any = 4;
- x = “hello”

void

```
function foo(): void {
  console.log(“foo");
}
```

null and undefined

- var x: string = null;
- var y:string= undefined

Defining New Types

- Type alias
  - Used to give a type a new name. It's like creating a shorthand for a more complex type.
- Interfaces
  - Used to define the shape of an object or the signature of a function. They are more extensible than type aliases because they can be reopened to add new properties.
- Classes
  - Define both the shape and the behavior of objects. A class can implement interfaces.
- Enums
  - Allow you to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Type aliases

- A type alias is declared using the ***type*** keyword followed by an identifier and a type annotation. Once defined, the alias can be used anywhere a type can be used.
- Flexibility: Type aliases can represent primitive types, unions, intersections, and any other valid TypeScript types.
- Readability: Using type aliases can make complex type definitions easier to work with and understand.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Interfaces

- Interfaces are a powerful way of defining contracts.
- Example

```
interface Vehicle{

    name: string;
    speed: number;
    gear?: number;

    applyBrakes(decrement: number): void;
}
```

- Interfaces can extend interfaces
  - (keyword extends)
- Classes implements interfaces
  - (keyword implements)

Classes

- Traditionally JavaScript uses functions and prototype-based inheritance to build up reusable components.
- Starting with ECMAScript 2015(ES6), JavaScript introduced the object-oriented class-based approach.
- Typescript supports classes that compile down JavaScript.
  - Works across all major browsers and platforms, without having to wait for the next version of the browser.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Classes

- Access Modifiers
  - public, private, protected
- Constructors
- Properties
- Static Members
- Inheritance
- Abstract classes and methods

Presentaed by Anil Joseph(anil.jos@gmail.com)

Arrow Functions

Represents a function expression

An arrow function expression has a shorter syntax than a function expression.

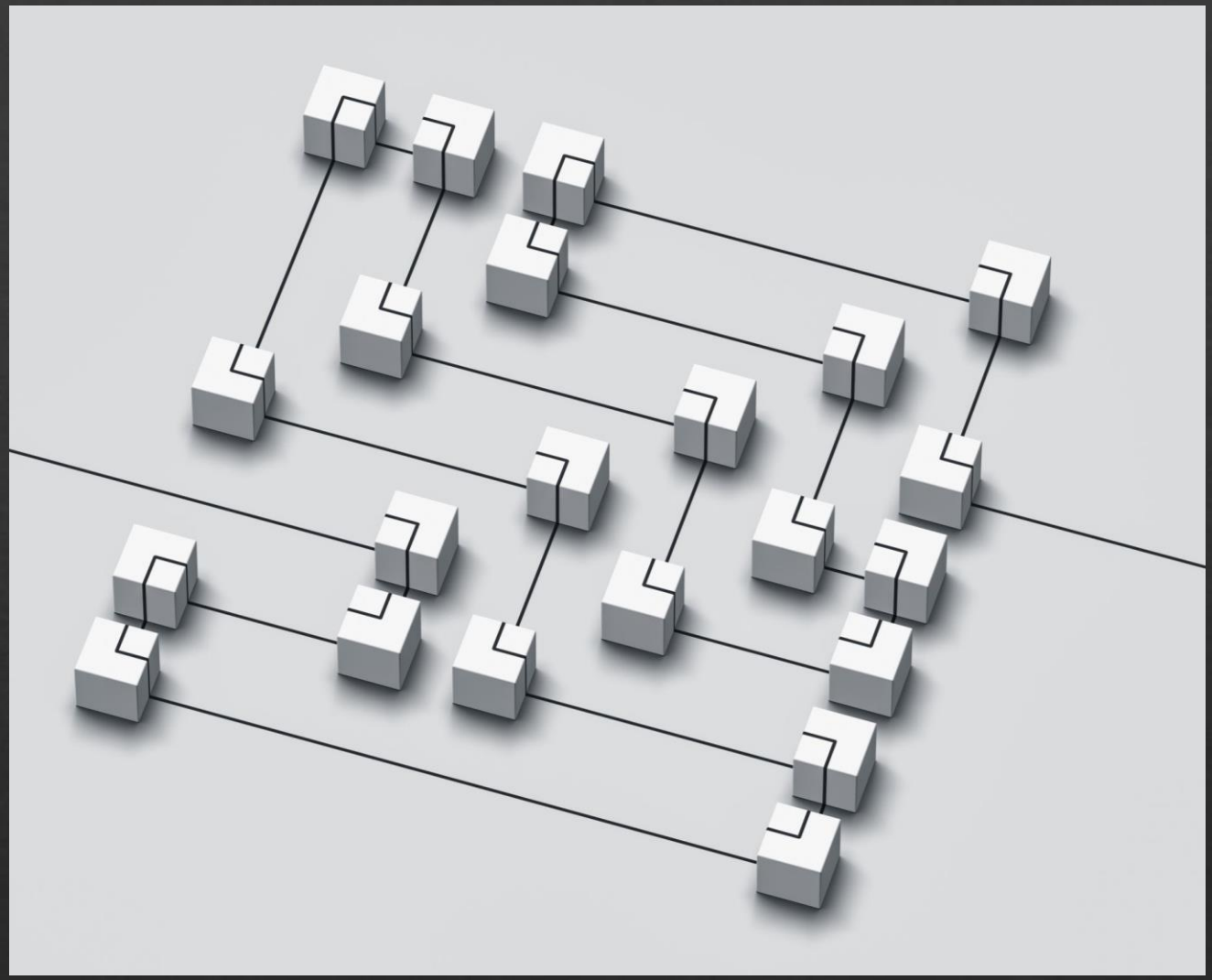
They do not receive the implicit arguments “this” and “arguments”.

Used widely for asynchronous and functional programming

Presentaed by Anil Joseph(anil.jos@gmail.com)

TypeScript Modules

- Starting with ECMAScript 2015(ES6), JavaScript has the concept of modules.
- Modules have a scope of their own
- In the module system every JS file is a module and all declarations in the file is scoped to that module.
- The same concept is shared in TypeScript



Presentaed by Anil Joseph(anil.jos@gmail.com)

Modules

- Use the import and export statements.

```
let foo = function(){
   //some code
}

export default foo;
```

one.js

```
import foo from ‘./one’;

foo();
```

two.js

```
import bar from ‘./one’;

bar();
```

three.js

Presentaed by Anil Joseph(anil.jos@gmail.com)

Modules

```
export let foo = function(){
   //some code
}

 export let bar = function(){
    //some code
 }
```

one.js

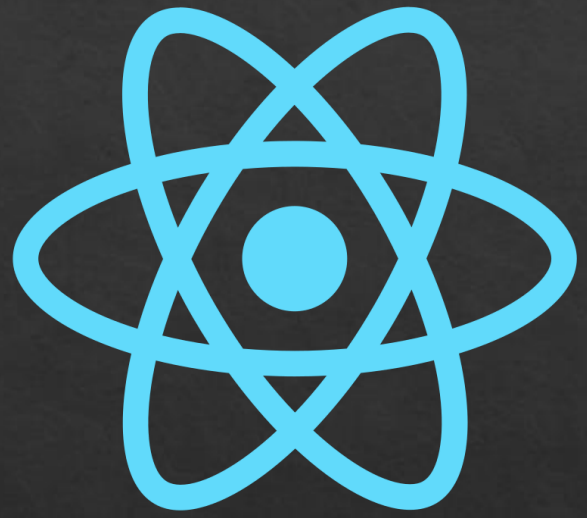
```
import {foo, bar} from './one';

foo();
bar();
```

two.js

Presentaed by Anil Joseph(anil.jos@gmail.com)

React

- JavaScript Library: React is a library for building user interfaces.
- Component-Based: UIs are built using reusable components.
- Declarative: React allows developers to write declarative code, making UIs predictable and easier to debug.
- Single-Page Applications (SPAs): Ideal for building interactive SPAs that load content dynamically without full page reloads.



Presentaed by Anil Joseph(anil.jos@gmail.com)

Core Features

- **JSX (JavaScript XML):**
  - Combines JavaScript and HTML-like syntax to make UI structure more readable.
  - Allows embedding JavaScript logic within HTML-like markup.
- **Component-Based Architecture:**
  - UIs are split into reusable components, each managing its own state and logic.
  - Components can be nested, composed, and reused throughout the app.
- **Virtual DOM**:
  - React uses a **Virtual DOM** to optimize performance.
  - Changes in the UI are first applied to the Virtual DOM, and only the differences (diffs) are applied to the real DOM.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Core Features

- Ecosystem Flexibility:
  - Can be used with other libraries or frameworks (like Redux for state management or Next.js for server-side rendering).
- Developer Experience:
  - Tools like React Developer Tools, JSX, and a clear component structure improve the overall development experience.
- SEO-Friendly (with SSR):
  - React can be combined with server-side rendering (SSR) using libraries like Next.js to improve SEO by rendering pages on the server.

Presentaed by Anil Joseph(anil.jos@gmail.com)

React History

- 2011: Origins
  - Developed by Jordan Walke, a software engineer at Facebook.
- 2013: Open Source
  - React was first released to the public at JSConf US in May 2013.
- 2014: React Grows
  - Instagram adopted React for their web app, marking a major milestone in React's real-world use.
- 2015: React Native
  - Launch of React Native, extending React’s core concepts to mobile app development for iOS and Android.
- 2017: React 16 (Fiber Release)
  - React 16 was released, powered by the Fiber architecture, improving the ability to handle updates efficiently and improving performance.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Getting Started

Two React Libraries

- React
- ReactDOM

JSX

- A JavaScript extension syntax allowing quoting of HTML

Babel

- The compiler for writing next generation JavaScript.
- Compiles JSX to JavaScript

Creating a React Project

Presentaed by Anil Joseph(anil.jos@gmail.com)

Creating a React Project

- Create and configure the project manually.
- Requires knowledge of multiple tools and their configuration
- Need to keep up with new versions and technologies
- Highly Customizable

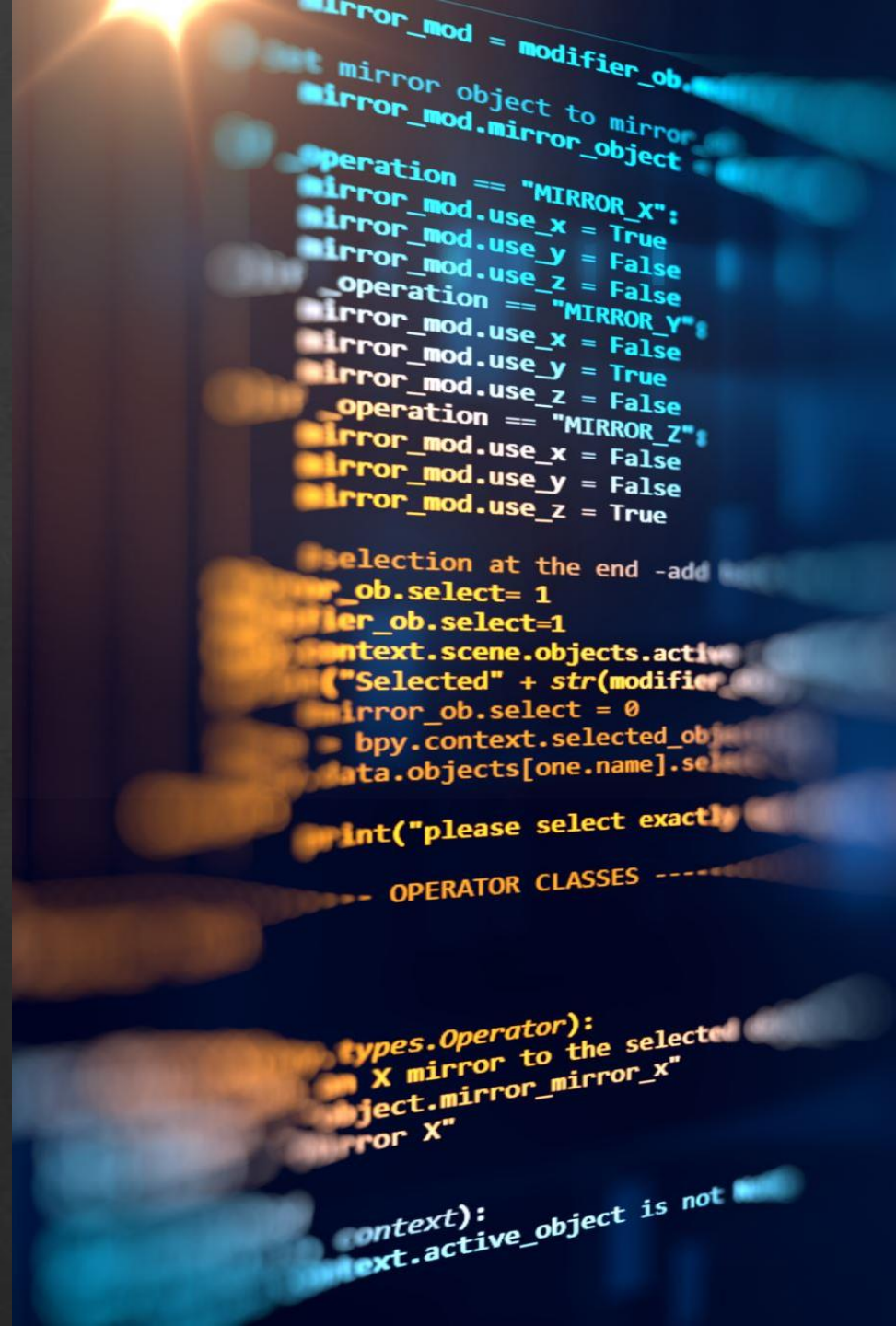
- Use a toolchain
- Easy to start with(Zero configuration)
- Customizations will be challenging

Presentaed by Anil Joseph(anil.jos@gmail.com)

Tool Chains

- A toolchain is a set of programming tools that is used to perform a complex software development task or to create a software product.(Wiki)
- Using a toolchain provides a better developer experience
- Advantages of Tool Chains
  - Scaling to many files and components.
  - Using third-party libraries from npm.
  - Detecting common mistakes early.
  - Live-editing CSS and JS in development.
  - Optimizing the output for production.

Presentaed by Anil Joseph(anil.jos@gmail.com)

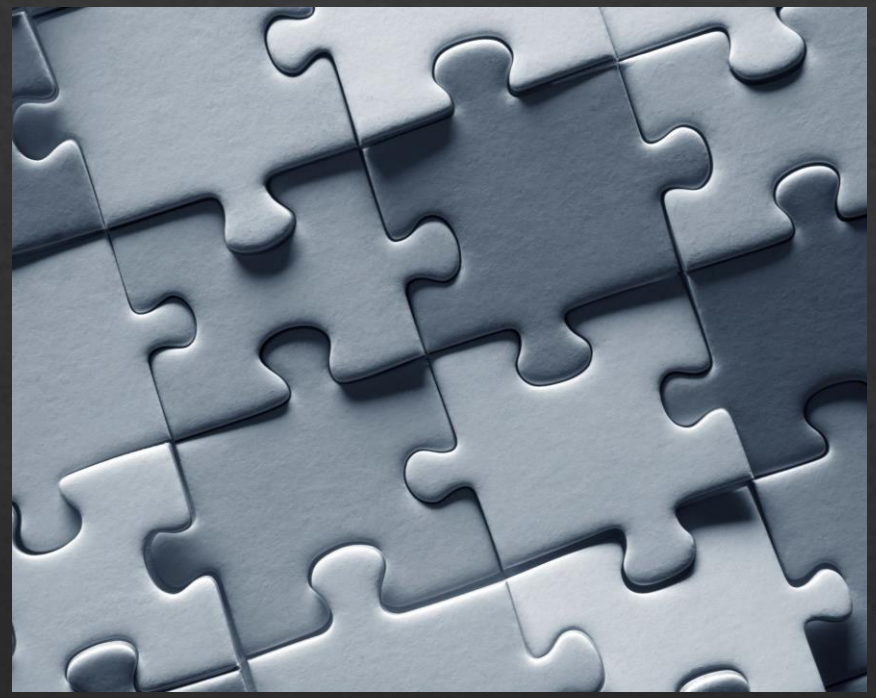


Popular Tool Chains

- Next.js
  - A React framework for building production-ready applications with features like server-side rendering (SSR), static site generation (SSG), and API routes.
- Create React App (CRA)
  - Officially maintained toolchain for quickly bootstrapping new React applications.
  - Provides a zero-config setup with support for JSX, ES6+, and development server out of the box.
- Remix
  - Popular React-based framework that emphasizes server-side rendering (SSR) and focuses on providing an optimized developer experience for full-stack applications.
- Vite
  - A fast build tool and development server, often used for React projects.
- React Native
  - Extends React for building mobile applications.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Create React App



- Create React App is an officially supported way to ***create single-page React applications***.
- It offers a modern build setup with no configuration.
- Sets up the development environment to use the latest JavaScript features.
- Optimizes the application for production.
- Integrates the tools: NPM, Babel, Webpack, Webpack development server

Presentaed by Anil Joseph(anil.jos@gmail.com)

Create Project

1

Install NodeJs

- Runtime to run/execute JavaScript on the machine/server
- This installation also installs npm(Node Package Manager)

2

Create React Application

- **npx create-react-app the-react-app**

3

Start the Application

- **cd the-react-app**
- **npm start**

Presentaed by Anil Joseph(anil.jos@gmail.com)

Next.js

- Next.js is a popular React framework
- Powerful features for building server-side rendered (SSR) and static web applications.
- Full stack development(JavaScript)

Presentaed by Anil Joseph(anil.jos@gmail.com)

Next.js Features

Full-Stack Framework	File-based Routing	Client-side Rendering (CSR)	Server-side Rendering (SSR)
Static Site Generation (SSG)	Incremental Static Regeneration (ISR)	Hybrid Rendering	API Routes
Automatic Code Splitting	Built-in CSS and Sass Support	Image Optimization	Fast Refresh

Presentaed by Anil Joseph(anil.jos@gmail.com)

Create a Next.js Project

1

Install NodeJs

- Runtime to run/execute JavaScript on the machine/server
- Use Node 18.8 or above for the latest version(Next 14.x)

2

Create Next Application

- npx create-next-app@latest
- npx create-next-app@14.2.7

3

Start the Application

- **Go to the project directory**
- **Run npm run dev**

Presentaed by Anil Joseph(anil.jos@gmail.com)

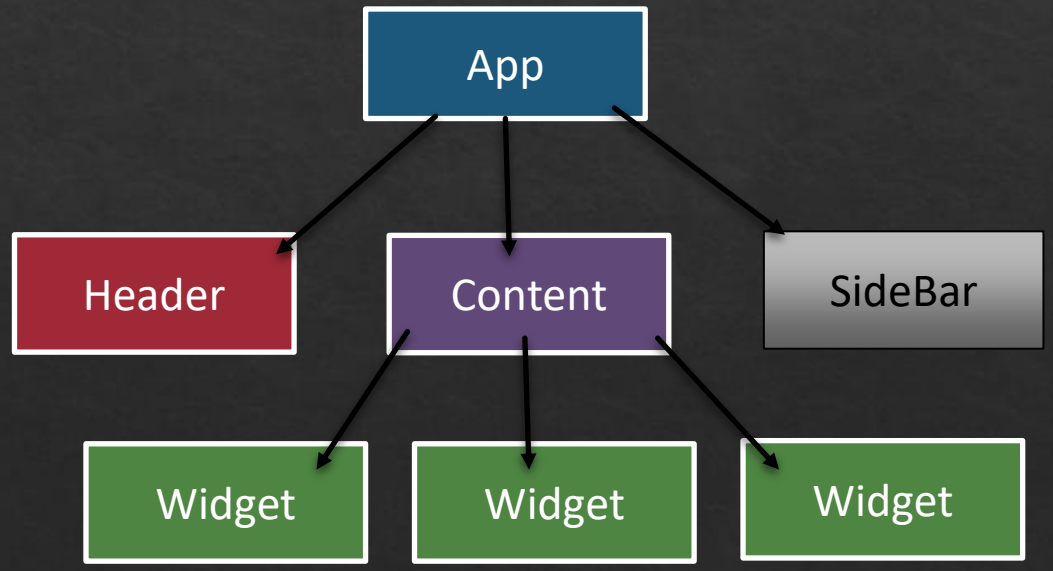
Next.js Project options

```
√ What is your project named? ... the-awesome-app
√ Would you like to use TypeScript? ... No / Yes
√ Would you like to use ESLint? ... No / Yes
√ Would you like to use Tailwind CSS? ... No / Yes
√ Would you like to use `src/` directory? ... No / Yes
√ Would you like to use App Router? (recommended) ... No / Yes
? Would you like to customize the default import alias (@/*)? » No / Yes
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

React Components

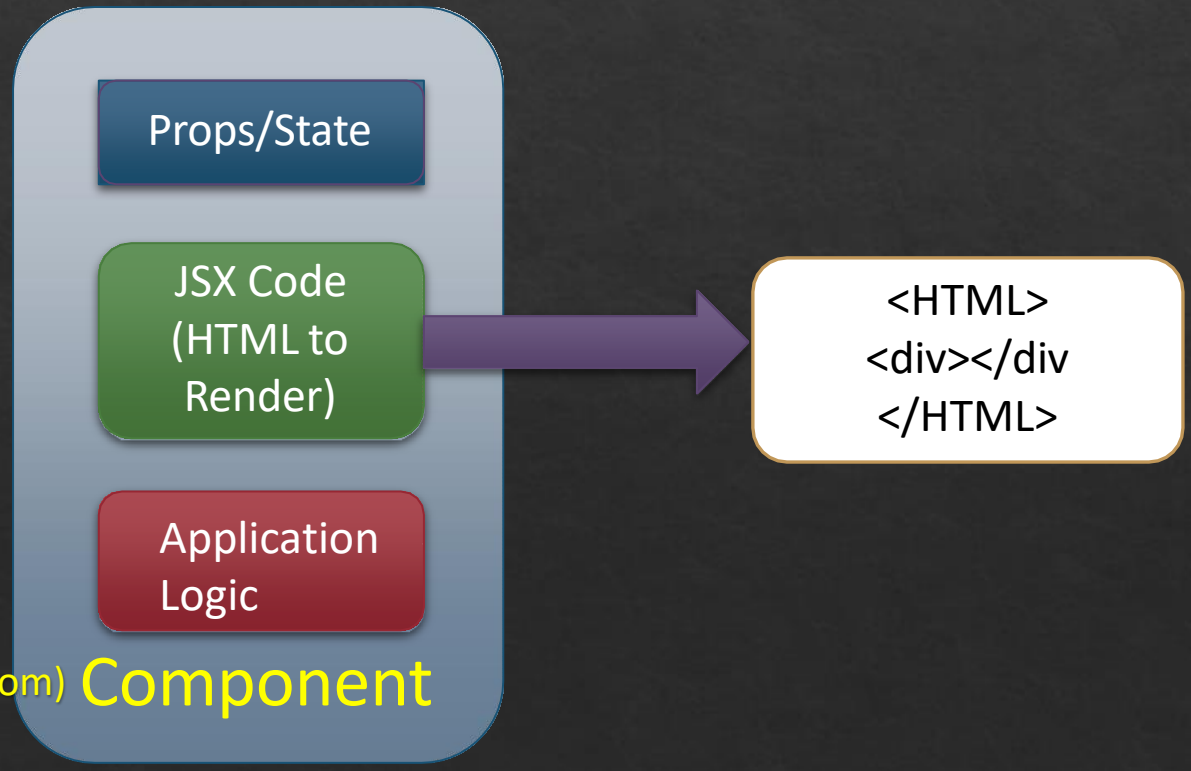
- Components are the core building blocks in React
- Creating a React applications is all about designing and implementing components
- A React application can be depicted as a component tree.



Presentaed by Anil Joseph(anil.jos@gmail.com)

React Components

- The UI in a React Application is composed of components, the building blocks.
- Components are designed to be reusable



Presentaed by Anil Joseph(anil.jos@gmail.com)

Types of Components

Functional

- Presentational
- Stateless (till React 16.8)
- Stateful (using React Hooks)

Class based

- Containers
- Stateful

Presentaed by Anil Joseph(anil.jos@gmail.com)

JSX

- *JSX* is a syntax extension for JavaScript.
- It was written to be used with React.
- JSX code looks a lot like HTML.
  - **It's actually JavaScript**
- A *JSX compiler* will translate any JSX into regular JavaScript.
- JSX elements are treated as JavaScript *expressions*.
  - They can go anywhere that JavaScript expressions can go.
- That means that a JSX element can be
  - Saved in a variable
  - Passed to a function
  - Stored in an object or array

Presentaed by Anil Joseph(anil.jos@gmail.com)

Components: Dynamic Content

- Dynamic content is outputted in the JSX using an expression.
- The syntax
  - { expression }
- This can be any one line expression
- Complex functionalities can be done by calling functions
  - { invokeSomeMethod() }

Presentaed by Anil Joseph(anil.jos@gmail.com)

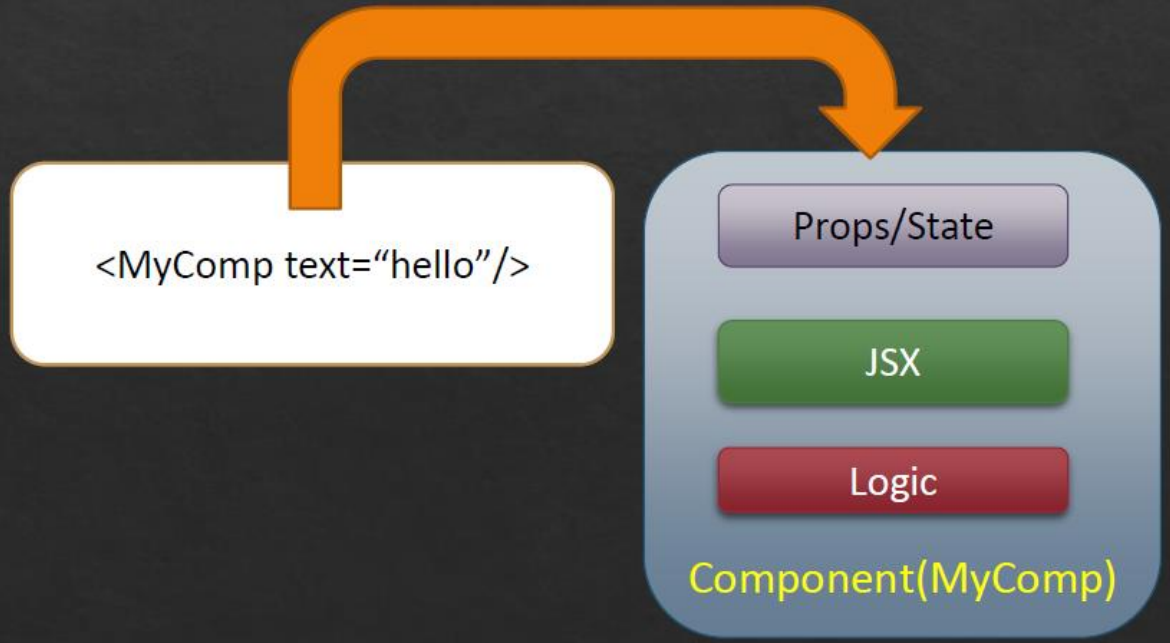
Components: Properties(props)

Most components can be customized with different parameters when they are created.

These creation parameters are called ***"props"***.

Props are used similar to HTML attributes.

Changes to props will automatically re-render the component.



Presentaed by Anil Joseph(anil.jos@gmail.com)

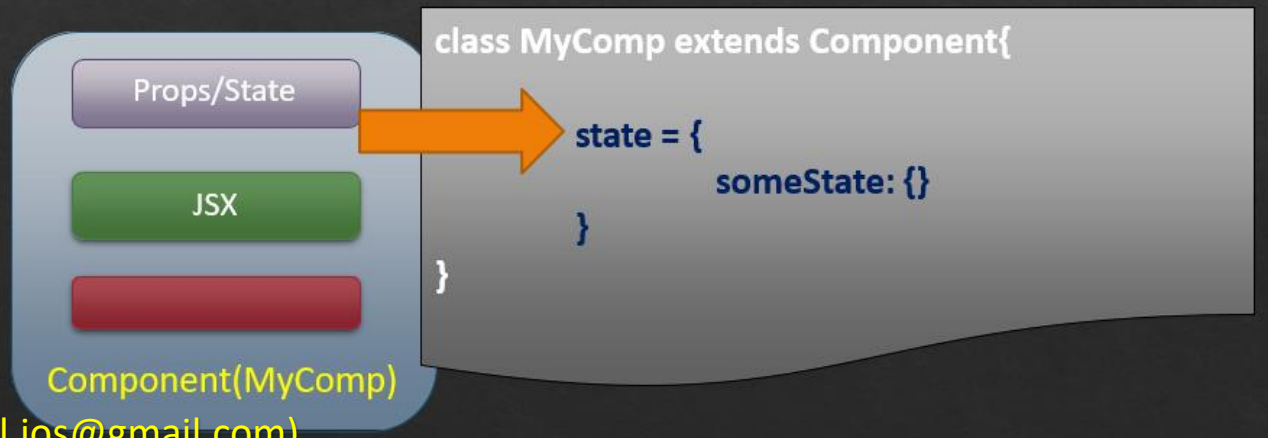
Component State

State holds information about the component

State is used when a component needs to keep track of information between renderings.

State is created and initialized in the component itself.

State updates trigger a rerender of the component.



Props and State

"props" and "state" are CORE concepts of React.

Only changes in "props" and/ or "state" trigger React to re-render the components and potentially update the DOM in the browser

props allow you to pass data from a parent (wrapping) component to a child (embedded) component.

State is used to change the component from within.

Event Handling

- Handling events with React elements is very similar to handling events on DOM elements with some syntactic differences.
- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.
- Event handlers will be passed instances of ***SyntheticEvent***
  - A cross-browser wrapper around the browser's native event
- Details
  - https://reactjs.org/docs/events.html

Component Lifecycle

React Hooks

React hooks was introduced in version 16.8

Many React features like state, lifecycle hooks etc. were available only with class-based components prior to 16.8.

Hooks let us use state and other React features in a functional component.

React team recommends the functional components over class-based since they can be highly optimized by the tools.

Presentaed by Anil Joseph(anil.jos@gmail.com)

React Hooks

State Hooks(useState)

- Equivalent of state in class-based components

Effect Hooks(useEffect)

- Used to perform side-effects in a function
- component
- Equivalent to lifecycle hooks in class-based components

Context Hooks(useContext)

- Used to access the React Context;

Presentaed by Anil Joseph(anil.jos@gmail.com)

React Hooks

Callback Hooks(useCallback)

- Returns a memoized callback.
- Used to optimize the components

Memo Hooks(useMemo)

- Returns a memoized value.
- Used to optimize the components

Ref Hooks(useRef)

- Returns a mutable ref object

Custom Hooks

- A functions
- Uses Other Hooks

Presentaed by Anil Joseph(anil.jos@gmail.com)

Components in Next.js 13

Client Components:

- Rendered and hydrated in the browser.

SSR Components:

- Server-Side Rendered and hydrated for interactivity.

Server Components:

- Rendered on the server with no client-side JavaScript.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Client Components

- Rendered on the client (browser) after initial page load.
- Can use React hooks like useState, useEffect, and event handlers (onClick).
- Suitable for interactive elements (e.g., buttons, forms).
- Require JavaScript on the client for interactivity.

Presentaed by Anil Joseph(anil.jos@gmail.com)

SSR Components (Server-Side Rendered Components)

- Rendered on the server for each request and sent to the client as fully rendered HTML.
- Uses getServerSideProps or data-fetching logic within the component.
- Interactive after hydration: Can use React hooks like useState, useEffect post-hydration.

Presentaed by Anil Joseph(anil.jos@gmail.com)

React Server Components

- Server Components are a new feature introduced in React to improve performance by rendering parts of the UI on the server and sending it as static HTML to the client.
- No Client-Side JavaScript:
  - Server Components don’t ship any JavaScript to the client. This reduces bundle sizes and improves performance.
- Seamless Integration:
  - Server Components work alongside Client Components, providing a seamless developer experience within the same React tree.
- Improved Performance:
  - By offloading intensive tasks to the server, Server Components reduce the need for client-side processing, leading to faster load times and better scalability.

Presentaed by Anil Joseph(anil.jos@gmail.com)

React Server Components

- Data Fetching:
  - Server Components are ideal for fetching data from databases or APIs without exposing sensitive logic to the client.
- Rendering Heavy UIs:
  - Offload rendering that requires heavy computation or multiple data sources to the server.
- Automatic Caching:
  - Server Components enable automatic caching and reduce unnecessary re-renders by separating the UI from client-side interactivity.

Presentaed by Anil Joseph(anil.jos@gmail.com)

React Server Components: Limitations

- No Interactivity:
  - Server Components cannot handle client-side interactions, like event listeners.
  - Cannot use hooks and other client specific features like Redux, React Context etc.
- Limited Access to Browser APIs:
  - Server Components run on the server, so they cannot directly interact with browser-specific APIs (e.g., window, localStorage).

Presentaed by Anil Joseph(anil.jos@gmail.com)

Client vs. Server Components

- Rendering Environment
  - Client Components: Rendered in the browser on the client-side.
  - Server Components: Rendered on the server and sent as static HTML to the client.
- Interactivity
  - Client Components: Can include interactive elements like event listeners (onClick, onChange), hooks (useState, useEffect), and browser-specific APIs (e.g., localStorage, window, document).
  - Server Components: Cannot include interactivity or use browser-specific APIs. They are designed to handle static data fetching and rendering without any client-side logic.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Client vs. Server Components

- Data Fetching
  - Client Components: Fetch data on the client-side using APIs such as fetch, axios, or any other client-side fetching method. They can only access client-side environment variables.
  - Server Components: Fetch data on the server-side before rendering the component. They have access to both client and server environment variables, and can perform secure server-side operations (like database queries, authentication checks, etc.).
- Rendering Strategy
  - Client Components: Are hydrated after the initial HTML is sent from the server. This means JavaScript code is sent to the browser and executed to make the component interactive.
  - Server Components: Render HTML on the server, and the client receives static HTML. No JavaScript is sent to the client for hydration unless a Client Component is nested inside.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Client vs. Server Components

- Performance
  - Client Components: May impact performance since JavaScript needs to be downloaded, parsed, and executed on the client. Hydration adds additional overhead.
  - Server Components: Better for performance because they only send minimal HTML to the client and don’t include client-side JavaScript unless needed. This leads to faster loading times.
- Bundle Size
  - Client Components: Increase the bundle size because JavaScript code needs to be sent to the client for execution.
  - Server Components: Reduce the bundle size since no client-side JavaScript is needed (except for interactive parts), resulting in a smaller footprint and quicker load times.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Client vs. Server Components

Feature	Client Component	Server Component
Rendered in	Browser (client-side)	Server
Interactivity	Supports interactivity	No interactivity
Data Fetching	Client-side	Server-side
Bundle Size	Increases bundle size	Minimal bundle size
Reactivity	Can use state and effects	No reactivity or state
Hooks	Can use all React hooks	Can only use server hooks (`use`)
Access to Browser APIs	Yes	No
Usage	Forms, dynamic UI elements	Static content, SEO pages
Third-party libraries	Can use client-side libraries	Limited to server-side

Presentaed by Anil Joseph(anil.jos@gmail.com)

Server-Side Rendering(SSR)

- Server-Side Rendering (SSR) is the process where the HTML for a webpage is generated on the server for each request, rather than in the browser.
- SSR provides better SEO and faster initial page load times, especially for content-heavy or data-driven applications.

- Pages in the Page Router can opt-in to SSR by exporting a special getServerSideProps function.

Presentaed by Anil Joseph(anil.jos@gmail.com)

SSR: Advantages

- Dynamic Data:
  - Pages that rely on data fetched at request time (e.g., user-specific data, time-sensitive content).
- SEO-Critical Pages:
  - Pages that need to be SEO-friendly, like blogs, marketing pages, or e-commerce product pages.Performance for
- First Load:
  - When you want the content to be available immediately on page load, without waiting for client-side JavaScript.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Static Site Generation (SSG)

- Static Site Generation (SSG) is a pre-rendering method where pages are generated at build time and served as static HTML files.
- SSG enables fast, highly-performant pages by serving pre-built static HTML with minimal server-side processing.
- Pages using SSG are generated at build time, meaning the content is static and does not change with each request.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Static Site Generation (SSG)

- Static Content:
  - Use SSG for pages where content doesn’t change often or can be updated on a scheduled basis (e.g., blogs, marketing pages, documentation).
- High Traffic Pages:
  - SSG is ideal for pages that receive high traffic because it serves lightweight static HTML.
- SEO-Critical Pages:
  - Similar to SSR, SSG is excellent for SEO as it produces fully rendered HTML for search engines to index.

Presentaed by Anil Joseph(anil.jos@gmail.com)

SSR & SSG in App Router

- In the **App Router** of Next.js, whether a page is statically generated using **Static Site Generation (SSG)** or server-rendered using **Server-Side Rendering (SSR)** depends on how you fetch and handle data in your components.
- Default Behavior:
  - In the App Router, ***components are statically generated by default*** if no asynchronous data fetching occurs during runtime. These components are pre-rendered at build time and served as static HTML files.
- When SSG is Used:
  - The component does not rely on dynamic data fetched during the request.
  - You are fetching static data at build time using generateStaticParams (for dynamic routes).

Presentaed by Anil Joseph(anil.jos@gmail.com)

SSG in App Router

```
export default function HomePage() {
  return (
    <div>
      <h1>Static Home Page</h1>
      <p>This page is statically generated at build time.</p>
    </div>
  );
}
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

SSG in App Router

```
export default async function HomePage() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return (
    <div>
      <h1>SSG with Next.js App Router</h1>
      <p>{data.message}</p>
    </div>
  );
}
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

SSG in Pages Router

```
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return {
    props: { data }, // Props passed to the page component
  };
}

export default function HomePage({ data }) {
  return (
    <div>
      <h1>Static Site Generated Page</h1>
      <p>{data.message}</p>
    </div>
  );
}
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

SSR in App Router

- SSR is Triggered:
  - If your component includes asynchronous data fetching or uses dynamic data that is fetched on each request, the App Router will opt for SSR. This ensures that fresh data is rendered for every request.
  - The caching is disabled on the fetch

Presentaed by Anil Joseph(anil.jos@gmail.com)

SSR in App Router

```
export default async function DashboardPage() {
  const res = await fetch('https://api.example.com/user-data', {
    cache: 'no-store', // Ensures fresh data is fetched on every request
  });
  const data = await res.json();

  return (
    <div>
      <h1>User Dashboard</h1>
      <p>Welcome, {data.name}</p>
    </div>
  );
}
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

SSR in App Router

```
export default async function PostPage({ params }) {
  const res = await fetch(`https://api.example.com/posts/${params.id}`);
  const post = await res.json();

  return (
    <div>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </div>
  );
}
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

Server-side rendering(Pages Router)

```
// pages/index.js
export async function getServerSideProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return {
    props: { data }, // Props passed to the page component
  };
}

export default function HomePage({ data }) {
  return (
    <div>
      <h1>Server-Side Rendered Page</h1>
      <p>{data.message}</p>
    </div>
  );
}
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

Incremental Static Regeneration (ISR)

- Incremental Static Regeneration (ISR) allows you to update static pages after they've been built, without needing to rebuild the entire site.
- Like SSG, ISR initially generates static HTML files at build time for each page.
- These files are served to users with pre-fetched data, making the site very fast.
- With ISR, you can specify a revalidate interval (in seconds), after which the page will be regenerated in the background the next time it is requested.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Incremental Static Regeneration in App Router

```tsx
export const revalidate = 60; // Revalidate every 60 seconds

export default async function PostsPage() {
  // Fetch data from an API
  const res = await fetch('https://api.example.com/posts');
  const posts = await res.json();

  return (
    <div>
      <h1>Latest Posts</h1>
      {posts.map((post) => (
        <div key={post.id}>
          <h2>{post.title}</h2>
          <p>{post.content}</p>
        </div>
      ))}
    </div>
  );
}
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

Incremental Static Regeneration (ISR)

```
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return {
    props: { data }, // Data passed to the page
    revalidate: 10,  // Revalidate every 10 seconds
  };
}

export default function HomePage({ data }) {
  return (
    <div>
      <h1>ISR Example</h1>
      <p>{data.message}</p>
    </div>
  );
}
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

Suspense and Streaming

- Suspense in React allows you to handle asynchronous operations (like data fetching) by providing a mechanism to "pause" rendering until the data is available.
- It lets you display a fallback UI while waiting for asynchronous tasks (like API calls, code splitting, or loading external resources) to complete.
- When a component is wrapped in <Suspense>, React can "suspend" rendering until the data or resource required by that component is fully loaded.
- A fallback UI is shown during the loading process.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Suspense and Streaming

- Streaming is a concept in React that allows incremental rendering of HTML to the browser as it is generated on the server.
- This feature works hand-in-hand with Server-Side Rendering (SSR),
- Allows the browser to render parts of the page as soon as they are ready, rather than waiting for the entire page to be rendered on the server.
- Progressive Rendering:
  - Content is sent to the client piece by piece, so parts of the page are rendered in the browser while other parts are still being generated on the server.
- Improved Time to First Byte (TTFB):
  - Since streaming sends data in chunks, the browser starts rendering earlier, improving the perceived performance.
- Better User Experience: Instead of waiting for the entire server-side page to be generated, users can start interacting with parts of the page that are already loaded.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Suspense and Streaming

- Progressive Rendering:
  - Content is sent to the client piece by piece, so parts of the page are rendered in the browser while other parts are still being generated on the server.
- Improved Time to First Byte (TTFB):
  - Since streaming sends data in chunks, the browser starts rendering earlier, improving the perceived performance.
- Better User Experience:
  - Instead of waiting for the entire server-side page to be generated, users can start interacting with parts of the page that are already loaded.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Parallel Fetch in Server Components

- Parallel fetching in React Server Components (and in Next.js) refers to the ability to fetch multiple data sources concurrently in server components to improve performance.
- Reduce Latency
- Improves Performance

Presentaed by Anil Joseph(anil.jos@gmail.com)

Parallel Fetch in Server Components

```tsx
export default async function Dashboard() {
  // Initiate both fetches separately
  const userPromise = fetchUserData();
  const postsPromise = fetchPostsData();

  // Wait for both promises to resolve in parallel
  const [user, posts] = await Promise.all([userPromise, postsPromise]);

  return (
    <div>
      <h1>Welcome, {user.name}</h1>
      <h2>Your Posts:</h2>
      <ul>
        {posts.map((post: any) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

Presentaed by Anil Joseph (anil.jos@gmail.com)

Caching in Next.js

- The fetch api’s are extended by Next.js for caching.
- const response = await fetch('https://api.example.com/data', { cache: 'force-cache’});
- cache: 'force-cache’:
  - Tells Next.js to cache the fetched data and reuse it until revalidation is triggered.
- cache: 'no-store': Bypasses caching and fetches fresh data every time.
- Cache Invalidation
  - Time-based: Use expiration times (e.g., revalidate: 60 in Next.js ISR).

Presentaed by Anil Joseph(anil.jos@gmail.com)

Next.js Routing

Next.js provides file-based routing

- In this mechanism the structure of your app or pages directory defines the routes.

Simplified Route Management

- No need to manually configure routes.

Intuitive and Readable Structure

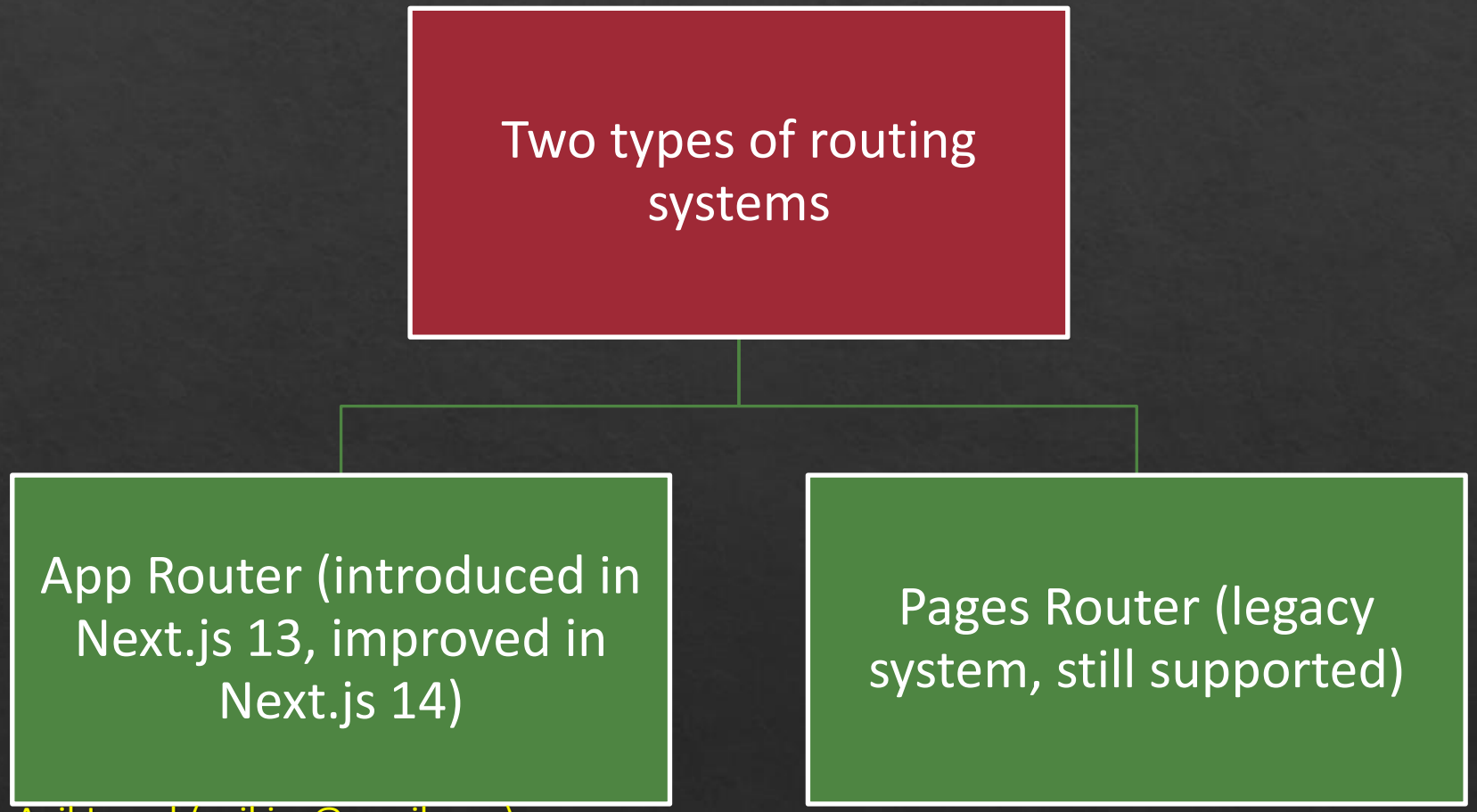
- The file and folder structure of your app visually represents the URL structure.

Dynamic Routing Simplified

- By using square brackets (e.g., [id].tsx), Next.js easily supports dynamic routes without additional configuration.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Next.js Routing



Presentaed by Anil Joseph(anil.jos@gmail.com)

Pages Router

- The Pages Router is the traditional routing system in Next.js, based on the pages directory.
- Data fetching is done using getStaticProps, getServerSideProps, or getStaticPaths.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Pages Router

```
pages/
├── api/                        # API routes folder
│   ├── users.ts                # API route for `/api/users`
│   └── products.ts             # API route for `/api/products`
├── dashboard/                  # Folder for `/dashboard` route
│   ├── index.tsx               # Renders `/dashboard` page
│   ├── analytics.tsx           # Renders `/dashboard/analytics`
│   ├── settings.tsx            # Renders `/dashboard/settings`
│   ├── error.tsx               # Custom error page for `/dashboard`
│   └── loading.tsx             # Custom loading state for `/dashboard`
├── _app.tsx                    # Custom App component for global layout and styles
├── _document.tsx               # Custom Document component for server-side logic
├── 404.tsx                     # Custom 404 page for unmatched routes
├── index.tsx                   # Homepage route (`/`)
└── sitemap.xml.ts              # Custom sitemap generation for SEO
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

App Router

- Introduced in Next.js 13, the App Router continues to evolve in Next.js 14.
- Provides powerful new features:
  - Server Components by default
  - Nested layouts and parallel routes
  - Data fetching at multiple levels (no need for getServerSideProps, getStaticProps)
  - Streaming and Suspense support

Presentaed by Anil Joseph(anil.jos@gmail.com)

App Router

```
app/
├── api/                          # API routes folder
│   ├── users/
│   │   └── route.ts              # API route for `/api/users`
│   └── products/
│       └── route.ts              # API route for `/api/products`
├── dashboard/                    # Folder for `/dashboard` route with nested layout
│   ├── layout.tsx                # Nested layout for all routes under `/dashboard`
│   ├── page.tsx                  # Page for `/dashboard`
│   ├── error.tsx                 # Error boundary for `/dashboard` routes
│   ├── loading.tsx               # Loading state for `/dashboard` routes
│   ├── analytics/
│   │   ├── page.tsx              # Nested route for `/dashboard/analytics`
│   │   ├── error.tsx             # Error boundary for `/dashboard/analytics`
│   │   └── loading.tsx           # Loading state for `/dashboard/analytics`
│   └── settings/
│       ├── layout.tsx            # Separate layout for `/dashboard/settings`
│       ├── page.tsx              # Page for `/dashboard/settings`
│       ├── error.tsx             # Error boundary for `/dashboard/settings`
│       └── loading.tsx           # Loading state for `/dashboard/settings`
├── error.tsx                     # Global error boundary for the app
├── global.css                    # Global styles applied across the app
├── layout.tsx                    # Root layout applied to the whole app
├── loading.tsx                   # Loading spinner for suspense or slow pages
├── not-found.tsx                 # 404 page for handling unmatched routes
├── page.tsx                      # Homepage route (`/`)
└── sitemap.ts                    # Custom sitemap generation for SEO
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

App Router API’s

- usePathname()
  - Returns the current route's pathname (e.g., /dashboard or /products/[id]).
  - Helps determine the current page and react to route changes in Client Components.
- useSearchParams()
  - Returns the current query parameters from the URL (e.g., /products?id=123&category=books).
  - Useful when working with query strings for filtering or dynamic data in Client Components.
- useRouter()
  - Provides methods like push(), replace(), and back() to navigate between routes.
  - Used to programmatically navigate between routes in Client Components.

Presentaed by Anil Joseph(anil.jos@gmail.com)

App Router API’s

- useParams()
  - Returns the dynamic route parameters (e.g., id in /products/[id]).

Presentaed by Anil Joseph(anil.jos@gmail.com)

API calls

- React does not provide any in-built library for API calls to the server.
- Common Libraries
  - Axios
  - Fetch
- Fetch is part of the W3C specifications and is supported by most modern browsers
- Axios is an open-source library based on the XMLHttpRequest(XHR) object with a lot of features.

Presentaed by Anil Joseph(anil.jos@gmail.com)

API Calls in Next.js

Next.js supports **both client-side and server-side** API calls.

You can fetch data in various ways depending on your needs

Client-Side:

- Fetching data in the browser after the page is loaded.

Server-Side:

- Fetching data on the server to send pre-rendered content to the client.

Static Site Generation (SSG):

- Fetching data at build time.

Next.js Backend API

- Next is a full stack framework.
- There a several ways to create and expose backend API's
- API routes
  - The traditional way, used with the pages router
- Route Handlers
  - Introduced in Next 13 replaces the API routes in the App router
- Server actions
  - Introduced in Next.js 13, a higher-level abstraction for interacting with the server without explicitly creating API routes

Presentaed by Anil Joseph(anil.jos@gmail.com)

Route Handlers

- Route handlers are used to define API endpoints in Next.js when using the App Router (introduced in Next.js 13).
- Serverless Architecture: Each API route is serverless and runs independently in the serverless environment (e.g., Vercel).
- Simplified API Management: No need for external backends, making full-stack development easier.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Route Handlers

- They live inside the app/api directory, and each file corresponds to a specific API endpoint.
- File-based routing
  - API routes are created by defining a route.ts or route.js file inside a folder representing the endpoint.
  - Example: app/api/products/route.ts defines the /api/products route.
- HTTP Method Handlers:
  - Route handlers can export functions that correspond to HTTP methods like GET, POST, PUT, DELETE.
- Dynamic Routes:
  - You can create dynamic routes by using square brackets in folder names, such as [id], allowing paths like /api/products/1.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Server Actions

- Server Actions in Next.js allow you to define actions that are executed only on the server. They enable server-side logic, such as data fetching or database operations, without exposing it to the client.
- Server Actions streamline handling sensitive tasks like interacting with databases, APIs, and session management directly on the server.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Server Actions

```tsx
export default function LoginPage() {
  async function handleLogin(formData: FormData) {
    'use server'; // This makes the function a Server Action
    const username = formData.get('username');
    const password = formData.get('password');

    // Example server-side operation (e.g., authentication)
    const res = await fetch('https://api.example.com/authenticate', {
      method: 'POST',
      body: JSON.stringify({ username, password }),
    });

    const result = await res.json();

    // Server-side logic for login, redirect, etc.
  }

  return (
    <form action={handleLogin}>
      <input type="text" name="username" placeholder="Username" required />
      <input type="password" name="password" placeholder="Password" required />
      <button type="submit">Login</button>
    </form>
  );
}
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

Server Actions

- Form Submissions:
  - Handle form submissions directly on the server, such as login, registration, or checkout processes.
- Database Operations:
  - Execute database reads/writes securely, ensuring the client cannot access sensitive queries.
- File Uploads:
  - Manage file uploads on the server securely without exposing logic to the client.
- Email Sending:
  - Trigger email sending operations directly from server actions.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Route Handlers vs Server Actions

Aspect	Server Actions	Route Handlers
Usage Location	Embedded in components/pages in the **App Router**	Defined in the `app/api/` directory as API endpoints
Primary Use Case	Handle server-side logic like form submissions, CRUD operations directly within components	Used to define API endpoints to handle various HTTP methods (GET, POST, etc.)
Example Usage	Handling authentication, database queries, form submissions	Building RESTful API endpoints for external or internal consumption
Accessibility	Only accessible within the component where they are defined	Accessible via HTTP requests, similar to traditional API routes
Client Interaction	Automatically triggered by the component (e.g., on form submit), reducing need for direct HTTP requests	Requires explicit API calls from the client, like `fetch()` or `axios`
Security	Logic is hidden from the client; no direct HTTP endpoints are exposed	Direct API endpoints can be accessed via HTTP requests, so they need to be protected (e.g., authentication)

Presentaed by Anil Joseph(anil.jos@gmail.com)

Route Handlers vs Server Actions

Typical Use Cases	Form submissions, database operations, background tasks handled on the server	RESTful APIs for CRUD operations, third-party integrations, or external clients
Server-Side or Client-Side	Always runs **server-side** and cannot be accessed from the client directly	Runs **server-side** but is explicitly triggered by client HTTP requests
Caching	Controlled by the server-side component with options to cache or fetch fresh data	API route handlers have more explicit control over caching and HTTP headers
Setup	Embedded directly in components with `'use server'` directive	Requires setting up the API structure in `app/api/` to handle requests

Presentaed by Anil Joseph(anil.jos@gmail.com)

axios

- Promise based HTTP client for the browser and node.js
- Installation
  - npm install axios
- Features
  - Make http requests
  - Supports the Promise API
  - Intercept request and response
  - Transform request and response data
  - Cancel requests
  - Automatic transforms for JSON data
  - Client side support for protecting against XSRF

axios methods

axios.request({config})

axios.get(url, {config})

axios.post(url,{data}, {config})

axios.delete(url, {config})

axios.put(url,{data}, {config})

axios global defaults

Base URL

- axios.defaults.baseURL = 'https://abc.com’;

Headers

- axios.defaults.headers.common['Authentication'] = AUTH_TOKEN;

Headers for specific methods

- axios.defaults.headers.post['Content-Type'] = 'application/json';

Debugging

- Error Messages
  - Messages generated by React during development mode
- Browser Developer Tools
- React Tools
  - Tools for Chrome and Firefox
- Error Boundaries
  - Error boundaries are React components that **catch JavaScript errors anywhere in their child component tree**
  - **Log errors**
  - **Display a fallback UI**

Presentaed by Anil Joseph(anil.jos@gmail.com)

Virtual DOM

The Virtual DOM (VDOM)

- Where a “virtual”, representation of a UI is kept in memory
- This is synced with the “real” DOM.
- This process is called reconciliation.

Reconciliation

- When the render() function is called it creates a tree of React elements.
- On the next update render() will return a different tree
- React then figures out how to efficiently update the UI to match the most recent tree.
- Uses the Diff algorithm

Higher-order Components

- A Higher Order Component is just a React Component that wraps another one.
- Higher Order Components is a Pattern used extensively with React
- Uses
  - Code reuse, logic and bootstrap abstraction
  - Render Highjacking
  - State abstraction and manipulation
  - Props manipulation
- Its basically a functions that returns a class component with the Wrapped Component.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Single Page Applications

A single-page application is an application that works inside a browser and does not require page reloading during use.

SPA is fast, as most resources (HTML+CSS+Scripts) are only loaded once throughout the lifespan of application. Only data is transmitted back and forth.

Presentaed by Anil Joseph(anil.jos@gmail.com)

State Management

React Context

- Available from React 16.3

React Redux

- Library to manage state

Mob

- Library to manage state

RxJs

- Reactive Programming using Observables

Presentaed by Anil Joseph(anil.jos@gmail.com)

Types of State

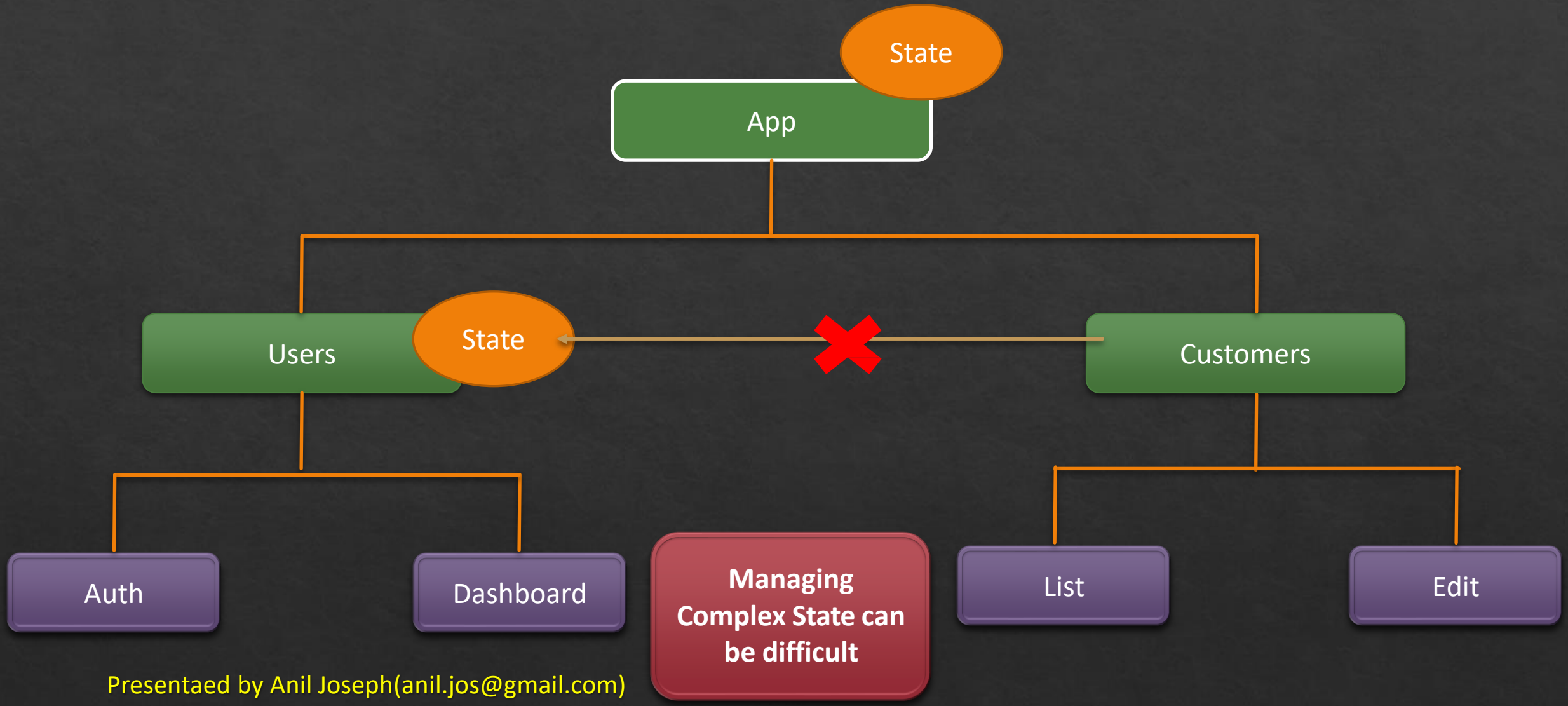
Redux

Redux is an open-source JavaScript library designed for managing application state.

It is primarily used together with React or Angular for building user interfaces.

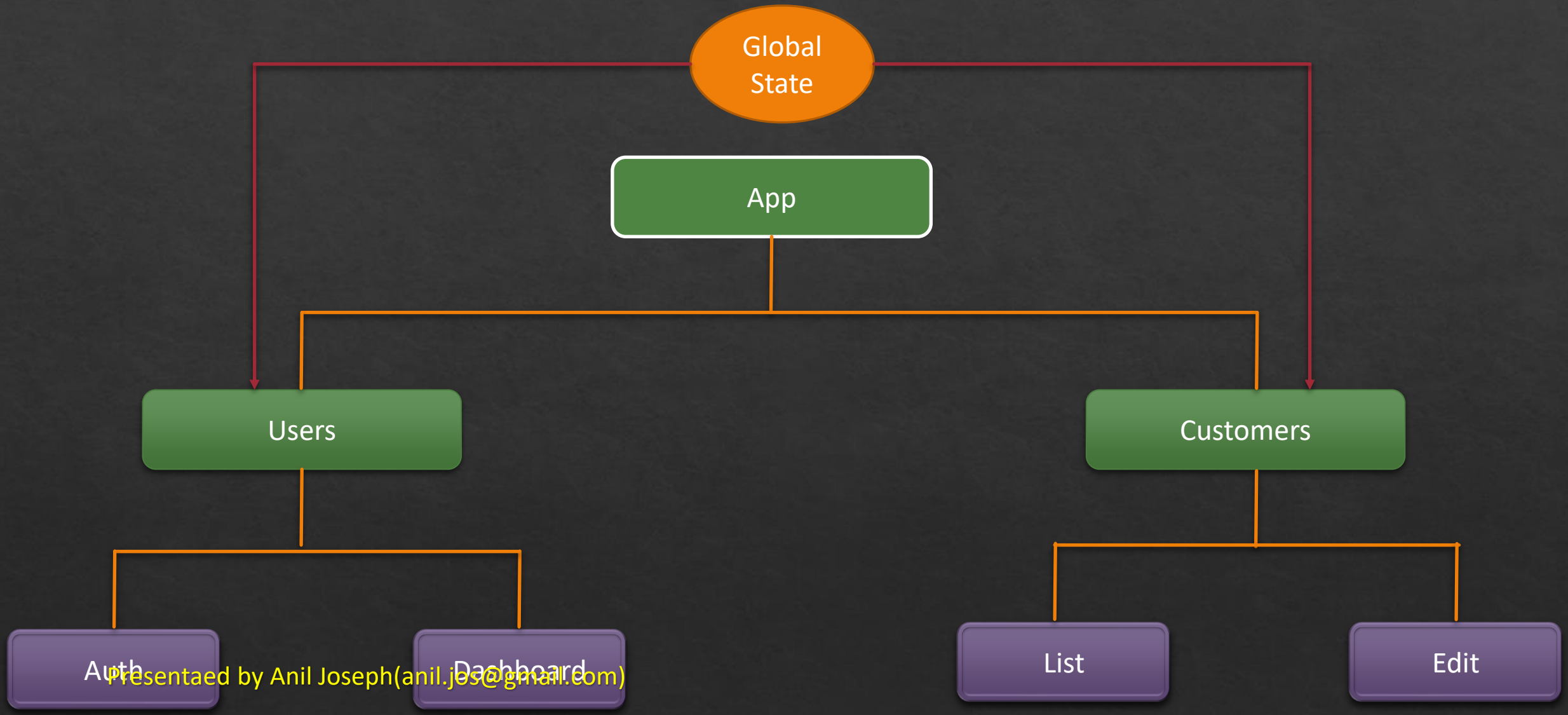
Redux was built on top of functional programming concepts.

Why Redux?

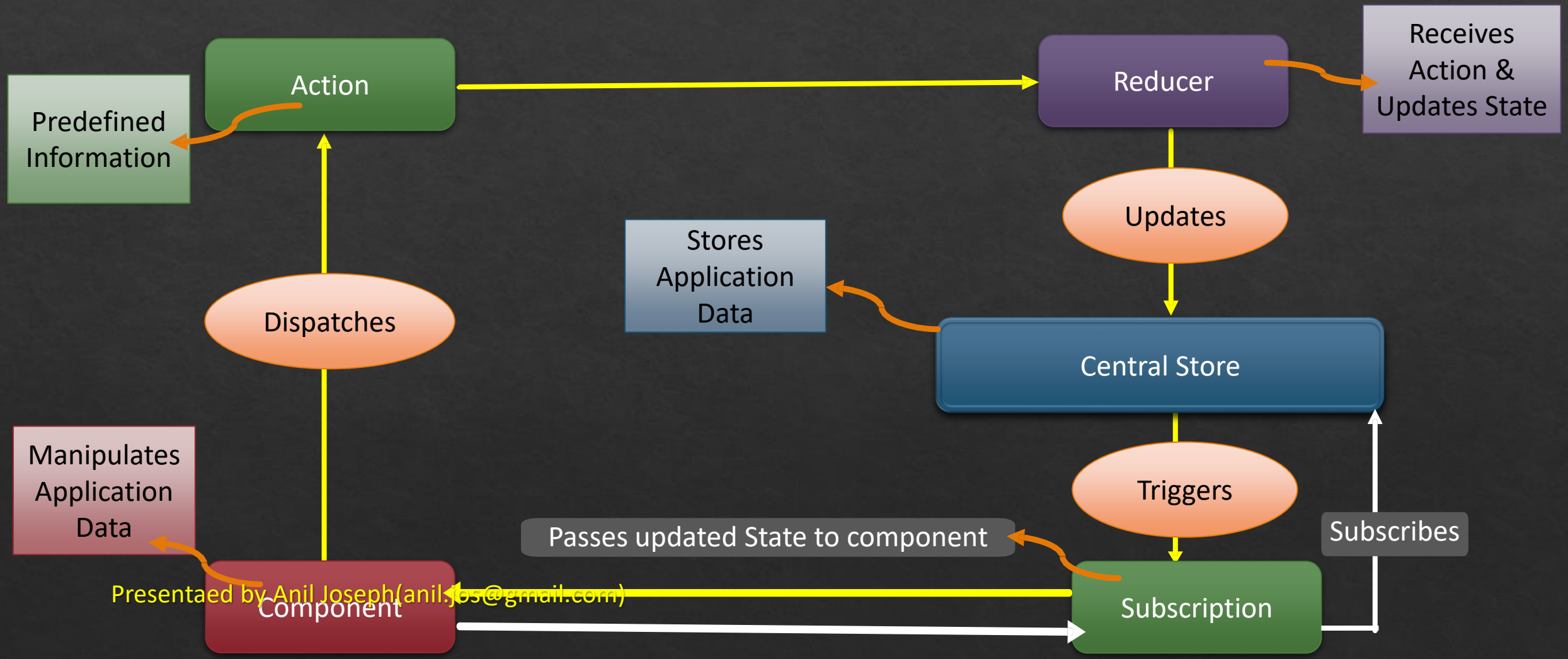


Presentaed by Anil Joseph(anil.jos@gmail.com)

Why Redux?

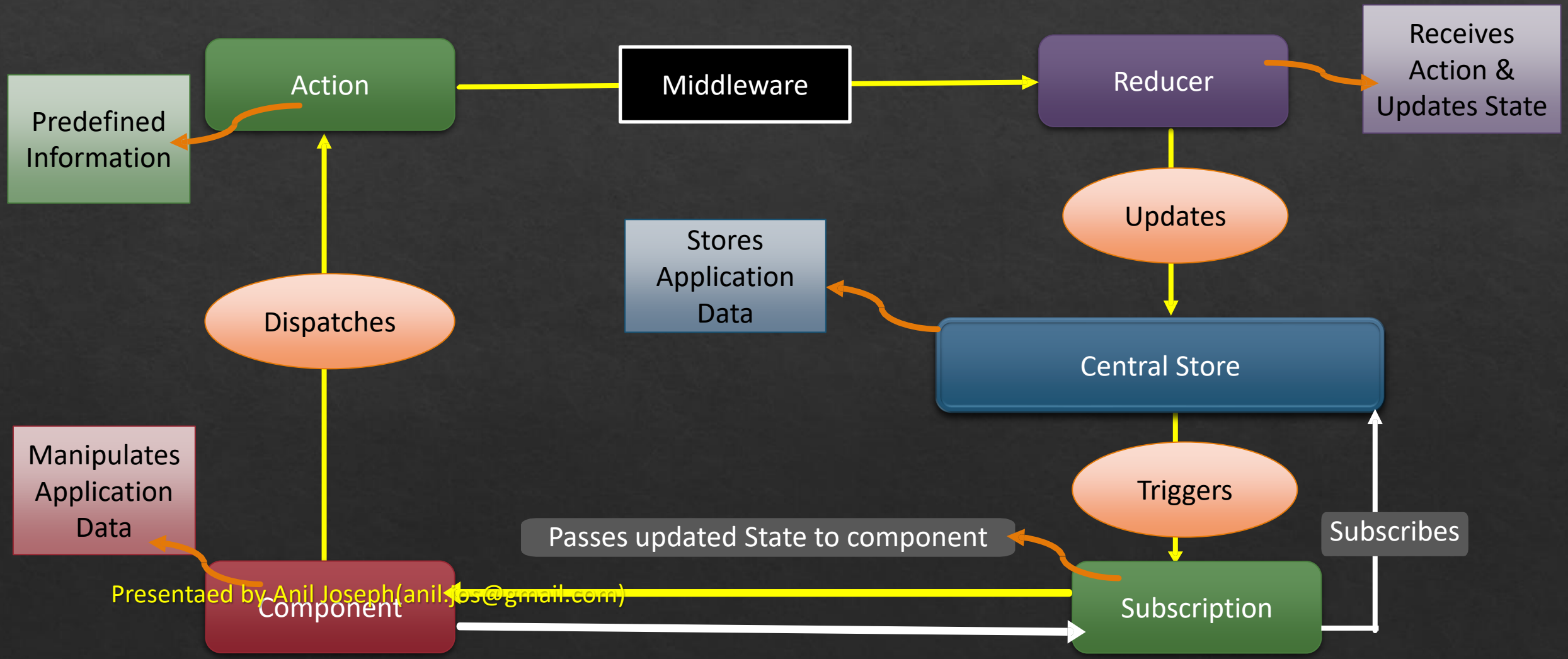


Redux Flow



Presentaed by Anil Joseph(anil.jos@gmail.com)

Redux Flow



Presentaed by Anil Joseph(anil.jos@gmail.com)

Redux Middleware

- Middleware in Redux provides an extension point between dispatching an action and the moment it reaches the reducer.
- Enables side effects, such as async operations, logging, and routing.
- Enhances functionality without modifying the core Redux workflow.

Presentaed by Anil Joseph(anil.jos@gmail.com)

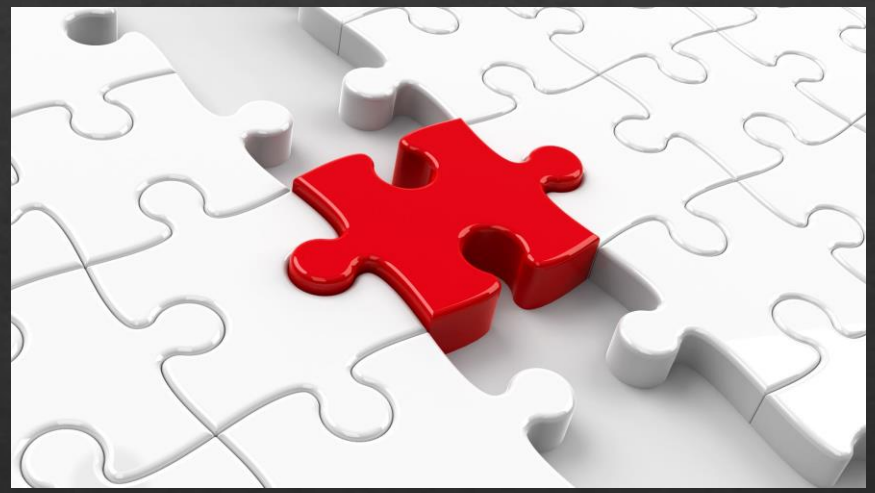
Redux Middleware

- Redux Thunk: Allows action creators to return a function instead of an action, enabling asynchronous logic.
- Redux Saga: Manages side effects with generator functions, providing more powerful async capabilities.
- Redux Logger: Logs actions and state changes for debugging purposes.

Presentaed by Anil Joseph(anil.jos@gmail.com)

React Redux

- Redux react is a library that integrates Redux to a React Application
- Comprises of Components & Functions
- Installation
  - npm install react-redux



Presentaed by Anil Joseph(anil.jos@gmail.com)

Redux Toolkit

- An official, opinionated, batteries-included toolset for efficient Redux development.
- Simplifies the process of writing Redux logic and reducing boilerplate code.
- Benefits
  - Reduced Boilerplate:
    - Provides a set of tools and functions that encapsulate common Redux patterns.
  - Improved Code Readability
    - Cleaner and more maintainable code.
  - Optimized Performance
    - Built-in support for creating efficient Redux logic.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Error Boundaries

- Error boundaries are React components that catch JavaScript errors anywhere in their child component tree
  - Used to log errors
  - Display a fallback UI
- Error boundaries do not catch errors for:
  - Event handlers
  - Asynchronous code
  - Server side rendering
  - Errors thrown in the error boundary itself
- A class component becomes an error boundary if it defines(either one)
  - componentDidCatch
  - static getDerivedStateFromError()

React Context

- Context provides a way to pass data through the component tree without having to pass props down manually at every level.
- Example of props to be passed down
  - Theme
  - Locale
  - Authenticated user
- API
  - React.createContext

Image

- Next.js <Image /> Component: A built-in component for optimized image loading.
- Automatically optimizes images by resizing, compressing, and serving the best format (e.g., WebP).
- Handles lazy loading out of the box, improving performance by loading images only when they enter the viewport.
- Can automatically generate different sizes for different screen resolutions.
- Next.js automatically compresses images, ensuring they are delivered in the most efficient format.
- The Next.js image component significantly reduces **Largest Contentful Paint (LCP)**, improving page speed scores.
- Image Sizes: Always specify width and height to prevent layout shifts and improve Core Web Vitals.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Code Splitting

- Code splitting allows you to split your app into separate bundles which your users can progressively load.
- API
  - React.lazy
  - Suspense

Automatic Code Splitting

- Next.js automatically splits your code by default, meaning each page only loads the JavaScript necessary for that page.
- Instead of loading the entire application at once, only the code for the current page and its dependencies are loaded, improving performance.
- You can manually split additional parts of your codebase using dynamic imports to load components only when needed.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Dynamic Imports for Additional Code Splitting

```
import dynamic from 'next/dynamic';

const DynamicComponent = dynamic(() => import('../components/HeavyComponent'));

export default function HomePage() {
  return (
    <div>
      <h1>Home Page</h1>
      <DynamicComponent />  {/* Loaded only when this part of the page is viewed */}
    </div>
  );
}
```

Presentaed by Anil Joseph(anil.jos@gmail.com)

Micro-Frontends

- Micro-frontends are an architectural style where a frontend application is decomposed into smaller, individual, and loosely coupled pieces, each owned by different teams.
- These pieces, or micro-frontends, can be developed, tested, and deployed independently, allowing for greater flexibility and scalability.
- Each micro-frontend is responsible for rendering a specific part of the user interface
- Each micro-frontend can be built using different technologies or frameworks, making the overall application more modular and easier to maintain.
- The approach extends the principles of microservices to the frontend, promoting better separation of concerns and enabling teams to work more autonomously.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Benefits of MFE

- Independent Deployments:
  - Each micro-frontend can be deployed independently, allowing for faster and more frequent releases without affecting the entire application.
- Improved Scalability:
  - Micro-frontends enable scaling individual parts of the application independently based on the traffic and load they handle.
- Technology Agnostic:
  - Different micro-frontends can be built using different technologies, enabling teams to choose the best tools and frameworks for their specific needs.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Benefits of MFE

- Better Team Collaboration:
  - Teams can work autonomously on their respective micro-frontends, reducing dependencies and improving productivity.
- Reduced Complexity:
  - By breaking down a large monolithic frontend into smaller pieces, the overall complexity of the application is reduced, making it easier to maintain and enhance.
- Enhanced User Experience:
  - Allows for incremental updates and A/B testing, leading to a more responsive and user-centric application.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Technologies to Create Micro-Frontends (MFE)

- Webpack Module Federation:
  - Enables multiple independently built and deployed bundles to form a single application.
  - Facilitates sharing of code and dependencies between different micro-frontends.
- Single SPA (Single Single Page Application):
  - A framework for building micro-frontends, allowing multiple frameworks to coexist in a single app.
  - Provides lifecycle hooks to load and unload micro-frontends.
- Piral
- Luigi

Presentaed by Anil Joseph(anil.jos@gmail.com)

Webpack Module Federation

- Webpack Module Federation is a feature introduced in Webpack 5
- Enables the sharing of modules between separate builds.
- Allows multiple independently built and deployed applications to dynamically share code and dependencies, forming a single cohesive application.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Key Features of Module Federation

- Dynamic Remotes:
  - Allows applications to load remote modules dynamically at runtime, rather than statically at build time.
- Shared Modules:
  - Facilitates sharing of common dependencies (e.g., React, lodash) between different micro-frontends, reducing duplication and ensuring consistency.
- Independent Deployment:
  - Enables each micro-frontend to be developed, tested, and deployed independently, making the development process more agile.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Key Features of Module Federation

- Version Management:
  - Handles different versions of shared libraries gracefully, allowing for compatibility between various parts of the application.
- Reduced Bundle Size:
  - By sharing common dependencies, Webpack Module Federation helps in reducing the overall bundle size, leading to faster load times.
- Seamless Integration:
  - Integrates seamlessly with existing Webpack configurations, making it easy to adopt without significant changes to the build setup.

Presentaed by Anil Joseph(anil.jos@gmail.com)

remoteEntry.js

- The file acts as the manifest that allows a host application to dynamically load and use modules exposed by a remote micro-frontend (MFE).
- Key Features
  - Metadata Provider: Contains metadata about the remote micro-frontend, including the modules it exposes, their locations, and the dependencies required.
  - Dynamic Loading: When the host application requests a module from the remote, the remoteEntry.js file provides the necessary information to locate and load that module at runtime.
  - Shared Dependencies: Includes information about shared dependencies, ensuring that both the host and remote use the same instance of shared libraries (e.g., React)

Presentaed by Anil Joseph(anil.jos@gmail.com)

remoteEntry.js

- When you build your remote MFE, Webpack generates the remoteEntry.js file based on the configuration specified in the ModuleFederationPlugin.
- The file contains references to all the exposed modules and the shared dependencies declared in the remote's Webpack configuration.
- When the host application starts, it fetches the remoteEntry.js file from the remote's server.
- This can be done dynamically at runtime, allowing the host to use the latest version of the remote without needing to rebuild.
- The host uses the metadata in remoteEntry.js to resolve and load the exposed modules dynamically.

Presentaed by Anil Joseph(anil.jos@gmail.com)

Next.js Multi Zones

- A feature in Next.js that allows you to compose multiple Next.js applications into a single site, with different "zones" running as separate Next.js apps.
- Allows you to break down a large app into multiple smaller, manageable apps, which can be deployed independently.
- Helps manage large applications with different responsibilities (e.g., blog, shop, user dashboard).

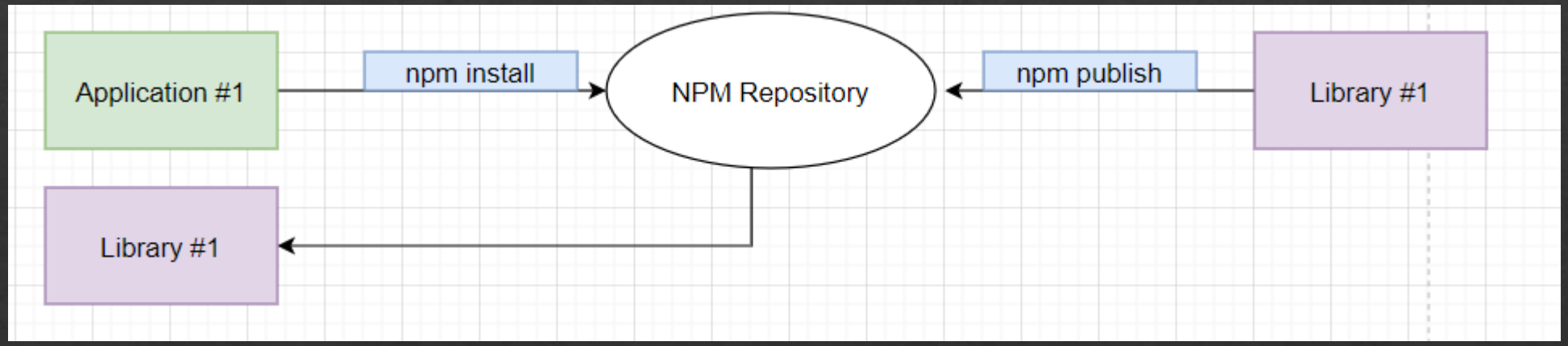
Presentaed by Anil Joseph(anil.jos@gmail.com)

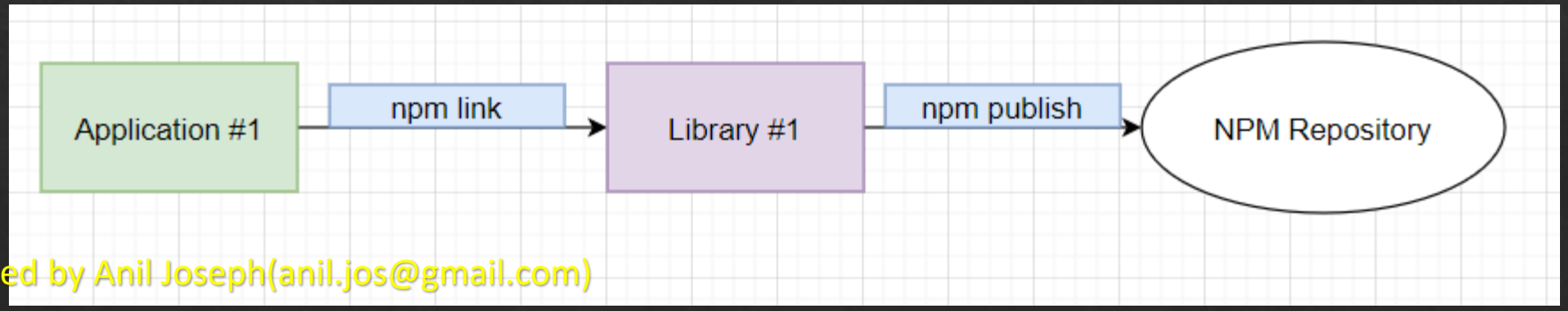
Design Patterns

- Higher Order Components
- Render Prop Pattern
- Provider
- Compound Component

Presentaed by Anil Joseph(anil.jos@gmail.com)

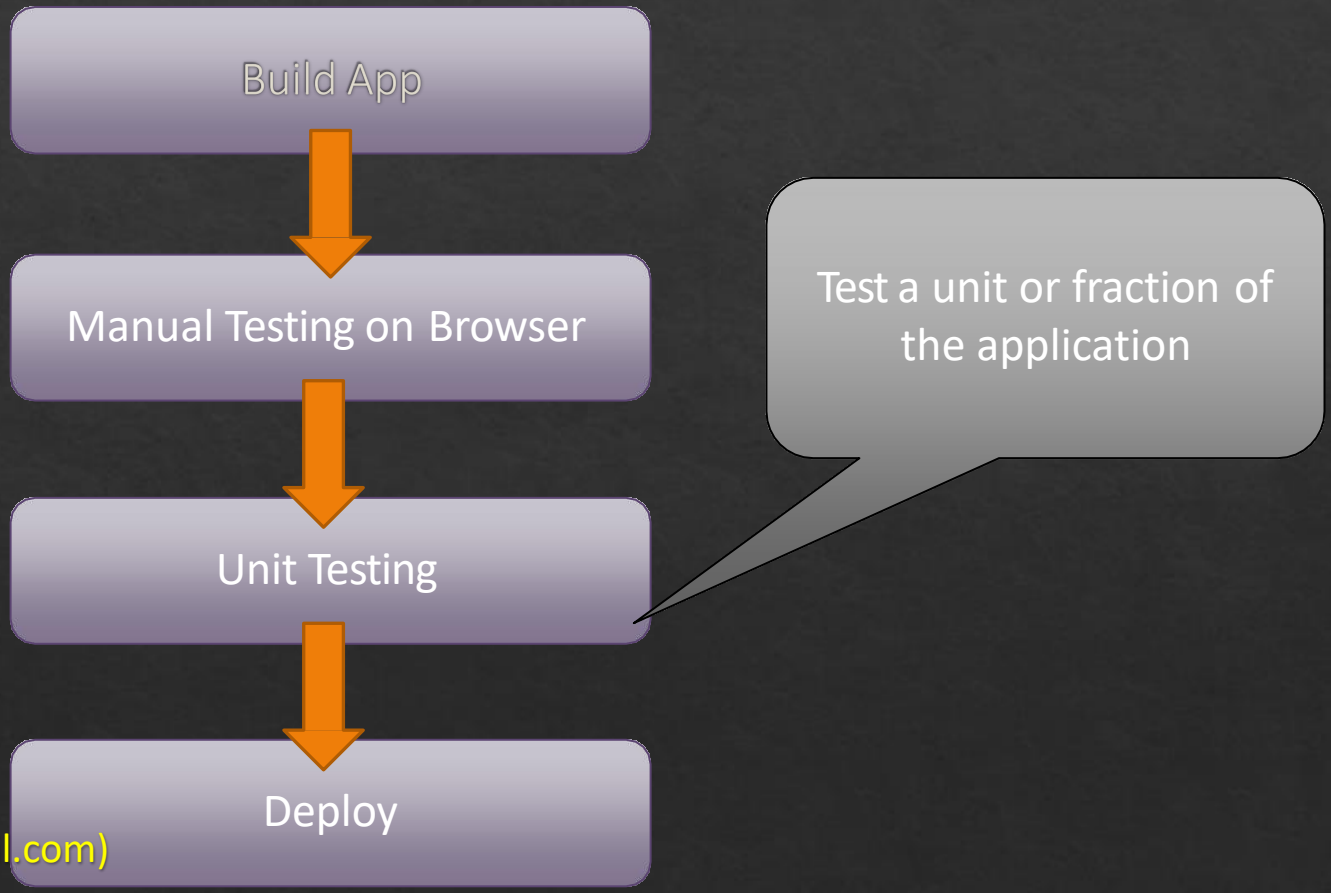
React Library





Presentaed by Anil Joseph(anil.jos@gmail.com)

Testing


Build App
Manual Testing on Browser
Unit Testing
Deploy
Test a unit or fraction of the application

Testing Tools

Testing API
&
Test Runner

Write the Unit Test.
Executes the Unit Tests.

Jest
(Built over Jasmine)

Test Utilities

Simulate the React App

react-test-renderer

enzyme

testing-library/react

Jest

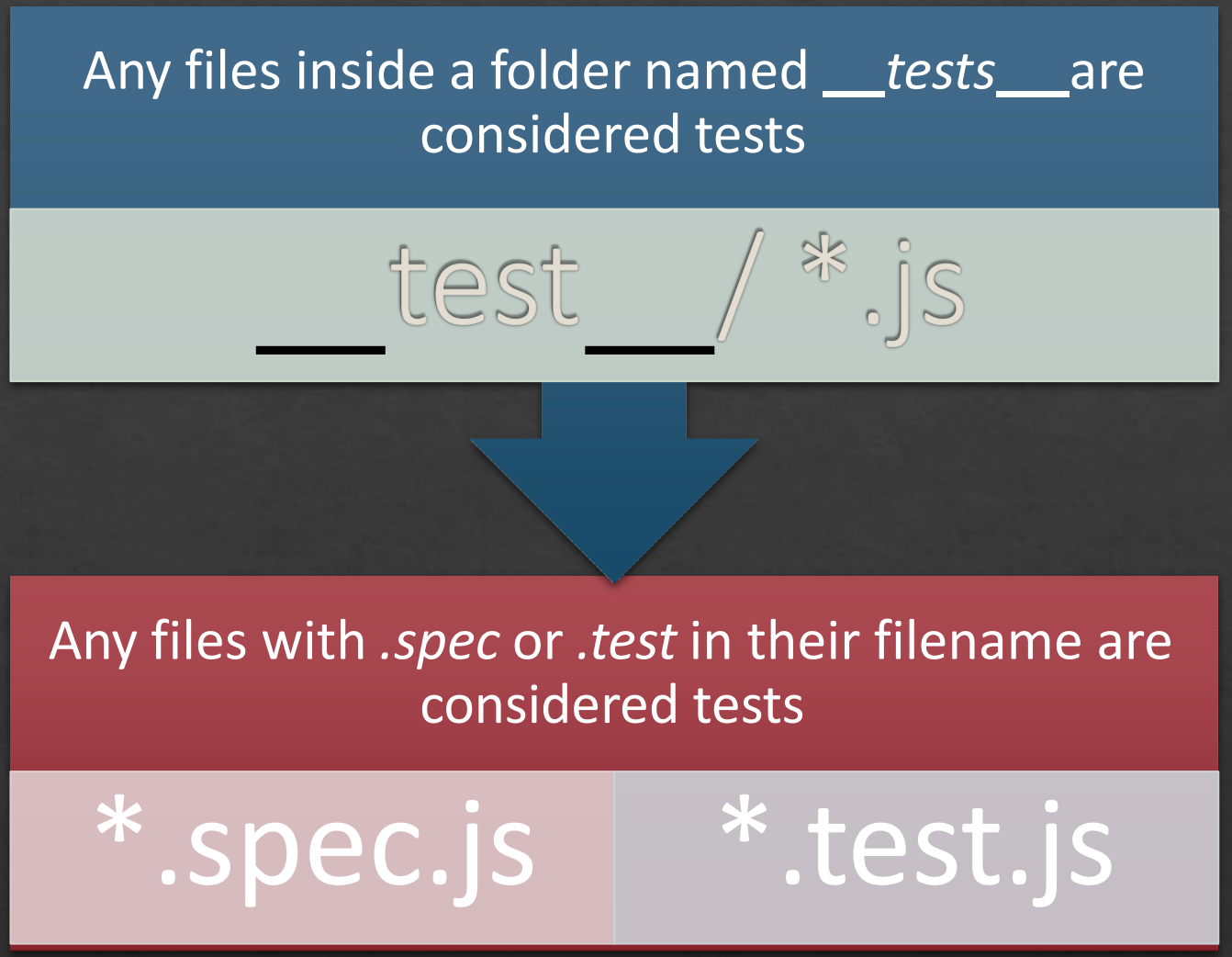
A testing library and test runner with handy features

Created by the members of the React team and the recommended tool for unit testing react

Built on top of Jasmine/Mocha

Additional features like mocking and snapshot testing

Jest: Identifying Test files



Jest Global Methods

it

- Method which you pass a function to, that function is executed as block of tests by the test runner.
- Alias name: test

describe

- An optional method for grouping any number of *it* or *test* statements
- Alias name: suite

Setup & Teardown Global functions

beforeEach	BeforeEach runs a block of code before each test
afterEach	Runs a block of code after each test
beforeAll	BeforeAll runs code just once, before the first test
afterAll	Runs a block of code after the last test)

Deployment

- Popular Platforms for Deploying Next.js
- Vercel (Recommended):
  - Official Next.js Deployment Platform.
  - Automatic support for SSR, SSG, Incremental Static Regeneration (ISR), and API routes.
  - Features: Automatic scaling, zero-config setup, serverless functions.
- Netlify:
  - Static site hosting with support for Next.js.
  - Automatic support for SSG, but requires configuration for SSR and API routes.
- AWS (Amazon Web Services):
  - Flexible and scalable cloud infrastructure.Use services like Lambda for serverless functions and S3 for static content.
- Other Platforms:
  - Heroku, DigitalOcean, Azure, Firebase Hosting, and GitHub Pages (for SSG).

Presentaed by Anil Joseph(anil.jos@gmail.com)

Thank You

ANIL JOSEPH

anil.jos@gmail.com