Full Stack Application Development

BITS Pilani

innovate achieve lead

BITS Pilani
Pilani Campus

Module 6: Understanding Frontend Development

Agenda

- ❑ Client Side Web APIs
- ❑ Need and Advantages of Frontend Frameworks
- ❑ Comparative study of frontend frameworks
- ❑ The Node Ecosystem
- ❑ Event Handing, Repaint and Reflow in Javascript

React - walkthrough

- ❑ React -Introduction
- ❑ React Components
- ❑ JSX
- ❑ Props and State
- ❑ Conditional Rendering
- ❑ React Hooks
- ❑ React Routing
- ❑ Redux

Back-end

Focus is on

- Software Architecture
- Application Business Logic
- Application Data Access
- Database management
- Scripting languages like JavaScript, Node.js, PHP, Python, Ruby, Java etc.
- Automated testing frameworks for the language being used
- Scalability
- High availability
- Security concerns, authentication and authorization

Front-end

Focus is on

- User Interface elements
- Mark-up and web languages such as HTML, CSS, JavaScript and supporting libraries
- Asynchronous request handling and AJAX
- Single-page applications (with frameworks like React, AngularJS or Vue.js)
- Web performance
- Responsive web design
- Cross-browser compatibility issues and workarounds
- End-to-end testing with a headless browser
- Build automation to transform and bundle JavaScript files, reduce images size
- Search engine optimization
- Accessibility concerns

Frontend

Elements

- Not just about the beautification of the web / mobile interfaces
- Involves elements for
  - Content Structure
  - Styling
  - Interaction
- Deals with
  - Rendering of the content on the browsers / within apps
  - Beautification of content rendered
  - Interaction carried out by user on web pages / mobile apps
- Building blocks – HTML + CSS + JS!

Content Structure

Markup

- Structure of the page is
  - the foundation of websites
  - essential for search engine optimization
  - vital to provide the style and the interaction that the reader will ultimately use

- Hyper Text Meta Language (HTML)
  - will be at the very center of it all regardless of how complex the site is
  - uses tags, as opposed to a programming language, in order to identify all the various types of content out there on every single webpage

  - For example, in a newspaper article, a header, sub header, text body and other things are present
  - HTML works in the exact same way to label all the stuff on the webpage, except it uses HTML tags
  - even a JavaScript webpage, is comprised of HTML tags corresponding to each element on the webpage and every single content type is bundled into HTML tags as well.

Styling

Cascading Style Sheets

- A core functionality of front-end development
- lays out the page and give it both its unique visual flair and a clear, user-friendly view to allow readers

- An important aspect of styling is checking across several browsers and to write concise, terse code that is
  - specific yet generic at the same time
  - displays well in as many renderers as possible

- CSS determines how all the HTML elements must appear on the frontend
  - HTML gives all of the raw tools necessary to structure webpage
  - CSS allows to style everything so it will appear to the user exactly the way you would like it to

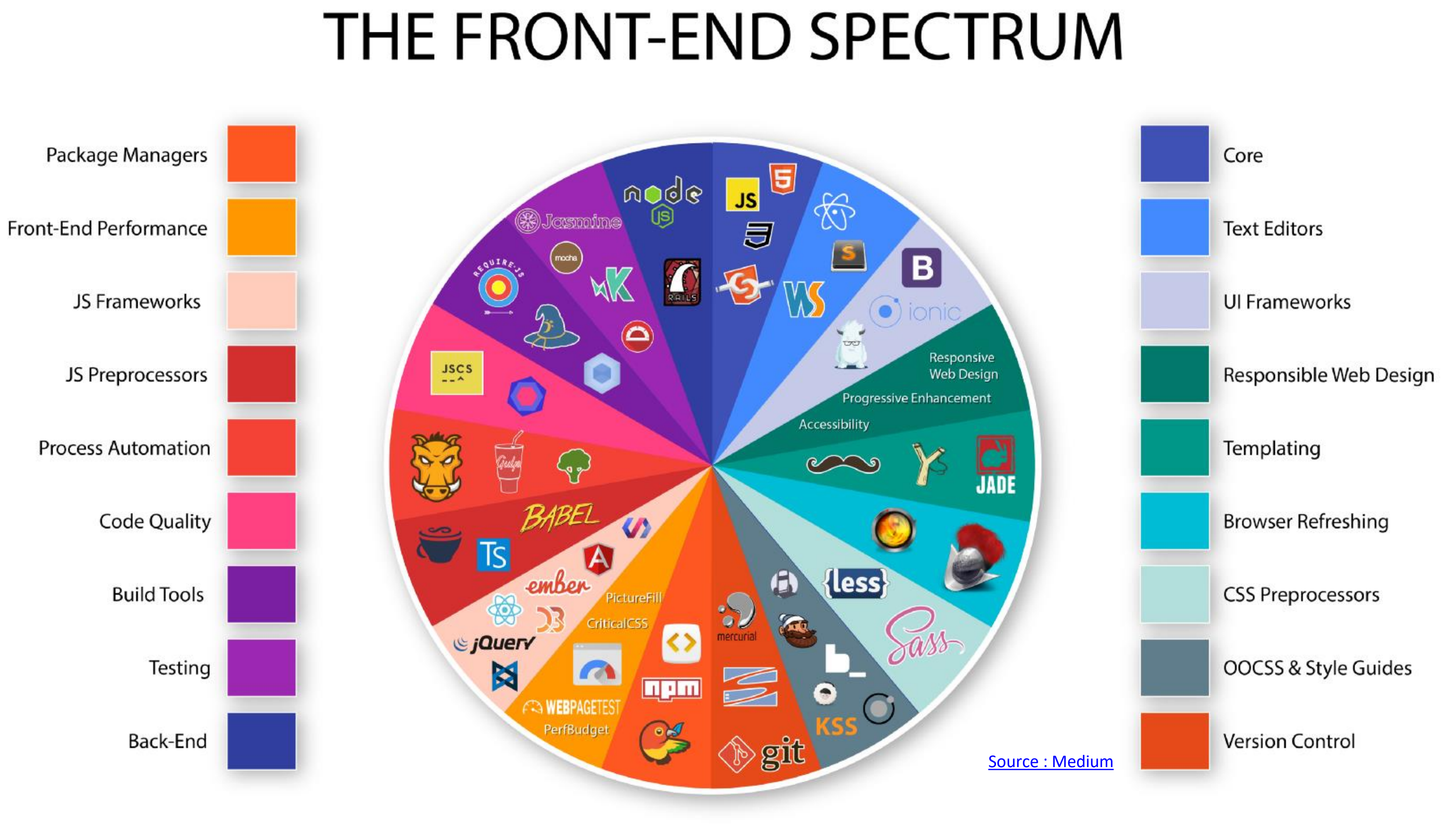
Interaction

User Interaction and interaction with backend

- JavaScript
- User Interaction on page
    - Supported by all modern browsers
    - employed by pretty much every website in order to gain increased functionality and power
    - Used mainly for website content adjustment and to make the website itself act certain ways depending on the user's actions
    - Used for creating call-to-action buttons, confirmation boxes and adding new details to current information

- **Dynamic Behavior**
    - drastically improves a browser's default actions and controls
    - helps in placing asynchronous requests to server side and render the response received

Frontend Tools Spectrum



Source : Medium

Web Platform

- Front-end technologies can run on the aforementioned operating systems and devices using the following run time web platform scenarios:
    - A web browser (examples: Chrome, IE, Safari, Firefox).
    - A headless browser (examples: phantomJS).
    - A WebView/browser tab
    - A native application

Web Platform

Web Browsers

- Is software used to retrieve, present, and traverse information on the WWW
- Typically run on a desktop or laptop computer, tablet, or phone,
  - but as of late a browser can be found on just about anything (i.e., on a fridge, in cars, etc.).

- The most common web browsers are (shown in order of most used first):
  - Chrome
  - Internet Explorer
  - Firefox
  - Safari

Web Platform

Headless Browsers

- Are a web browser without a graphical user interface t
- that can be controlled from a command line interface programmatically
- Used for the purpose of web page automation
  - e.g., functional testing, scraping, unit testing, etc.

- Think of headless browsers as a browser that can be run from the command line that can retrieve and traverse web pages

- The most common headless browsers are:
  - PhantomJS
  - slimerjs
  - trifleJS

Web Platform

Webviews

- Are used by a native OS, in a native application, to run web pages
- Think of a webview like an iframe or a single tab from a web browser that is embedded in a native application running on a device
    - e.g., webviews iOS, android, windows

- The most common solutions for webview development are:
    - Cordova (typically for native phone/tablet apps)
    - NW.js (typically used for desktop apps)
    - Electron (typically used for desktop apps)

Web Platform

Native from Web Tech

- Web browser development can be used by front-end developers to craft code for environments that are not fueled by a browser engine
- Development environments are being dreamed up that use web technologies (e.g., CSS and JavaScript), without web engines, to create native applications

- Some examples of these environments are:
  - NativeScript
  - React Native

Why frameworks?

- For frontend developers, it's increasingly challenging to make up their minds about which JavaScript application framework to choose
  - especially when need to build a single-page application

- Requirements
- First, modern frontend JavaScript frameworks must respect the Web Components specification.
  - should have support for custom HTML elements building
- Second, a solid JavaScript framework should have its own ecosystem
  - Ready solutions aim at solving various problems of client-side development such as
    - ❖ Routing
    - ❖ managing app state
    - ❖ communicating with the backend

Vue.js React ember

BACKBONE.JS ANGULAR

React

- Stormed the JS world several years ago to become its definite leader
- Encourages to use a reactive approach and a functional programming paradigm
- Introduced many of its own concepts to define its unique approach to frontend web development
- Need to master a component-based architecture, JSX, and unidirectional data flow

- Ecosystem:
  - The React library itself plus React-DOM for DOM manipulation
  - React-router for implementing routes
  - JSX, a special markup language that mixes HTML into JavaScript
  - React Create App, a command line interface that allows you to rapidly set up a React project
  - Axios and Redux-based libraries, JS libraries that let you organize communication with the backend.
  - React Developer Tools for Chrome and Firefox browsers.
  - React Native for development of native mobile applications for iOS and Android.

Angular 2

- Marks a turning point in the history of the Angular framework
- Has substantially changed its architecture to come to terms with React
- Is from a Model-View-Whatever architecture to a component-based architecture

- Ecosystem:
  - A series of modules that can be selectively installed for Angular projects: @angular/common, @angular/compiler, @angular/core, @angular/forms, @angular/http, @angular/platform-browser, @angular/platform-browser-dynamic, @angular/router, and @angular/upgrade
  - TypeScript and CoffeeScript, supersets of JavaScript that can be used with Angular
  - Angular command line interface for quick project setup
  - Zone.js, a JS library used to implement zones, otherwise called execution context, in Angular apps
  - RxJS and the Observable pattern for asynchronous communication with server-side apps
  - Angular Augury, a special Chrome extension used for debugging Angular apps
  - Angular Universal, a project aimed at creating server-side apps with Angular
  - NativeScript, a mobile framework for developing native mobile applications for iOS and Android platforms

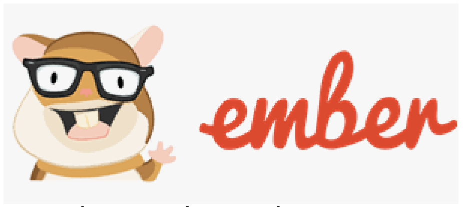
Vue

- At first sight, it may look like the Vue library is just a mix of Angular and React
- Borrowed concepts from Angular and React
- For example,
  - Vue wants you to store component logic and layouts along with stylesheets in one file. That's how React works without stylesheets
  - To let Vue components talk to each other, Vue uses the props and state objects - existed in React before Vue adopted it
  - Similarly to Angular, Vue wants you to mix HTML layouts with JavaScript
- The VueJS ecosystem consists of:
  - Vue as a view library.
  - Vue-loader for components.
  - Vuex, a dedicated library for managing application state with Vue; Vuex is close in concept to Flux.
  - Vue.js devtools for Chrome and Firefox.
  - Axios and vue-resource for communication between Vue and the backend.
  - Nuxt.js, a project tailored to creating server-side applications with Vue; Nuxt.js is basically a competitor to Angular Universal.
  - Weex, a JS library that supports Vue syntax and is used for mobile development.

Ember

- Like Backbone and AngularJS, is an “ancient” JavaScript framework
- But the fact that Ember is comparatively old doesn’t mean that it’s out of date
- Allows implement component-based applications just like Angular, React, and Vue do
- One of the most difficult JavaScript frameworks for frontend web development
- Realizes a typical MVC JavaScript framework, and Ember’s architecture comprises the following parts:
  - adapters, components, controllers, helpers, models, routes, services, templates, utils, and addons.

- The EmberJS ecosystem includes:
  - The actual Ember.js library and Ember Data, a data management library.
  - Ember server for development environments, built into the framework.
  - Handlebars, a template engine designed specifically for Ember applications.
  - QUnit, a testing JavaScript framework used with Ember.
  - Ember CLI, a highly advanced command line interface tool for quick prototyping and managing Ember dependencies.
  - Ember Inspector, a development tool for Chrome and Firefox.
  - Ember Observer, public storage where various addons are stored. Ember addons are used for Ember apps to implement generic functionalities.



Backbone/Marionette

- Is an MV* framework. Backbone partly implements an MVC architecture, as Backbone’s View part carries out the responsibilities of the Controller
- Has only one strong dependency – the Underscore library that gives us many helper functions for convenient cross-browser work with JavaScript
- Attempts to reduce complexity to avoid performance issues

BACKBONE.JS

- The BackboneJS ecosystem contains:
  - The Backbone library, which consists of events, models, collections, views, and router
  - Underscore.js, a JavaScript library with several dozen helper functions that you can use to write cross-browser JavaScript
  - Underscore’s microtemplating engine for Backbone templates
  - BackPlug, an online repository with a lot of ready solutions (Backbone plugins) tailored for Backbone-based apps
  - Backbone generator, a CLI for creating Backbone apps
  - Marionette, Thorax, and Chaplin – JS libraries that allow you to develop a more advanced architecture for Backbone apps

Compared

Feature	React	Angular	Ember	Vue
Architecture	Component-based	Component-based (MVC)	Component-based (MVC)	Component-based
Learning Curve	Relatively steep, especially for beginners	Steeper learning curve, especially for complex apps	Moderate	Easier compared to Angular, similar to React
Performance	Highly performant due to Virtual DOM	Generally, performs well, but can be slower in some cases	Good performance	Lightweight and fast, similar to React
Community Support	Large and active community	Massive community support	Active community	Growing community
Flexibility	Flexible, allows integration with other libraries	Comprehensive framework with built-in features	Opinionated, but customizable	Flexible and can be integrated with existing projects

Client Side Web API

Client-side web APIs are the tools that breathe life into web applications by allowing your JavaScript code to interact with various functionalities. There are two main categories:

1. **Browser APIs:** These APIs provide access to features and functionalities of the web browser itself, the device it's running on, and the operating system.
2. **Third-Party APIs:** These APIs are offered by external services and allow you to integrate their data or functionalities into your application. For instance, you could use a maps API to display an interactive map on your website or a social media API to enable users to log in with their existing accounts.

Client-Side Web API Examples

Browser API

DOM manipulation: The Document Object Model (DOM) API lets you interact with the structure and content of your webpages. You can dynamically add, remove, or modify HTML elements, change styles, and manipulate the overall layout.

Fetch API: This modern API provides a powerful and flexible way to make HTTP requests to servers and retrieve data. It simplifies asynchronous data fetching compared to older techniques like XMLHttpRequest.

Geolocation API: This API allows you to access the user's location information, if they grant permission. This can be useful for building location-based features like weather apps or finding nearby restaurants.

Canvas API: This API empowers you to create dynamic graphics and animations directly on the web page. It provides a powerful way to visualize data or create interactive games and experiences.

Web Storage API: This API offers various options for storing data locally on the user's device. Local Storage allows persistent data storage, while Session Storage only persists data for the current browser session. This can be useful for things like remembering user preferences or keeping track of application state.

window.localStorage window.SessionStorage

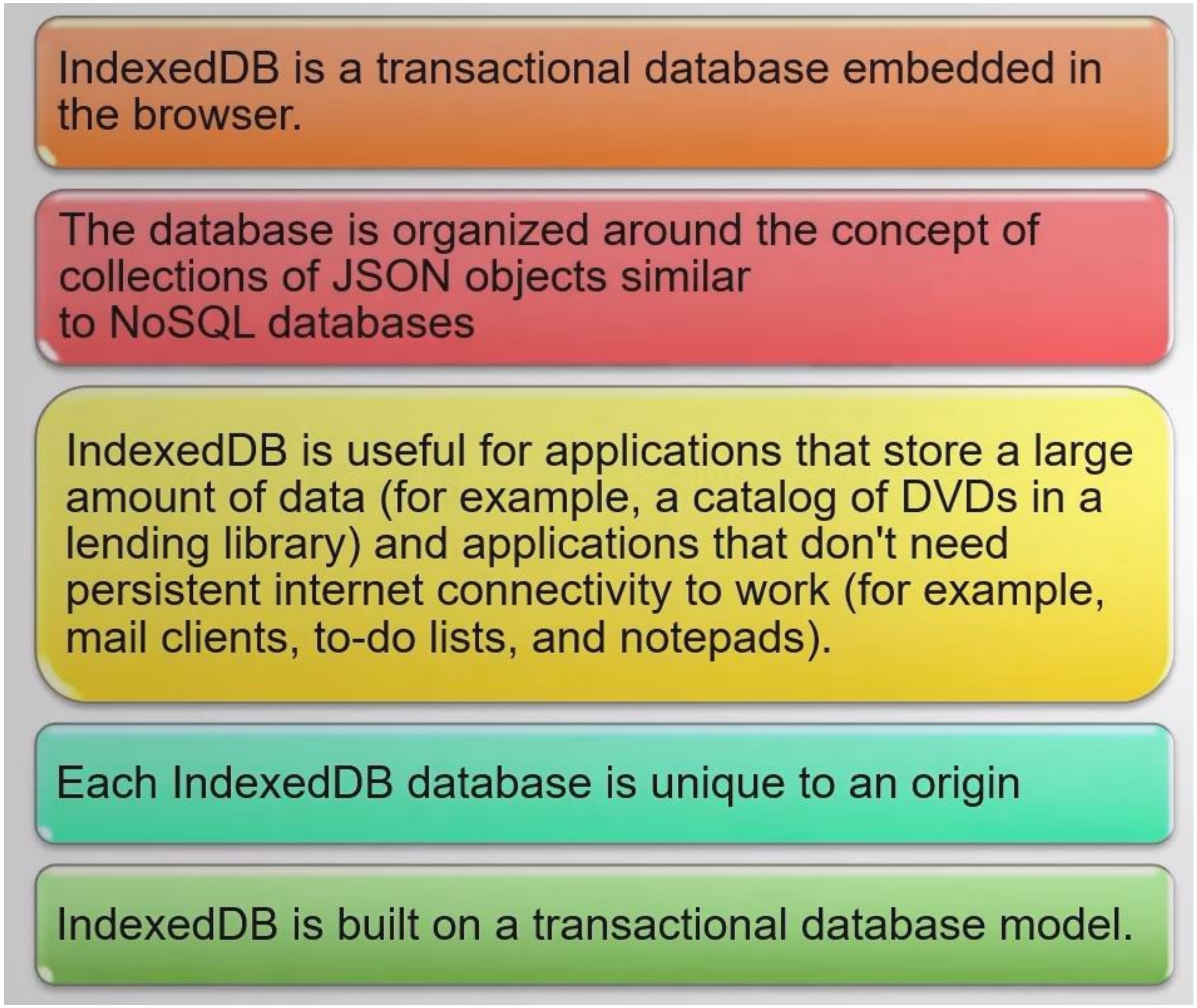
Client-side Web API Examples

Third-Party APIs:

- **Social Media APIs:** These APIs allow you to integrate social media features like logins, sharing content, or retrieving user information. (e.g., Facebook API, Twitter API)
- **Maps APIs:** These APIs provide map functionalities like displaying interactive maps, searching for locations, or getting directions. (e.g., Google Maps Platform, OpenStreetMap)
- **Payment APIs:** These APIs enable you to integrate payment processing functionalities into your application. (e.g., Stripe, PayPal)
- **Weather APIs:** These APIs provide access to weather data and forecasts for specific locations. (e.g., OpenWeatherMap, AccuWeather)

By effectively leveraging client-side web APIs, you can create dynamic, interactive, and feature-rich web applications that provide a compelling user experience.

Indexed DB



IndexedDB is a transactional database embedded in the browser.

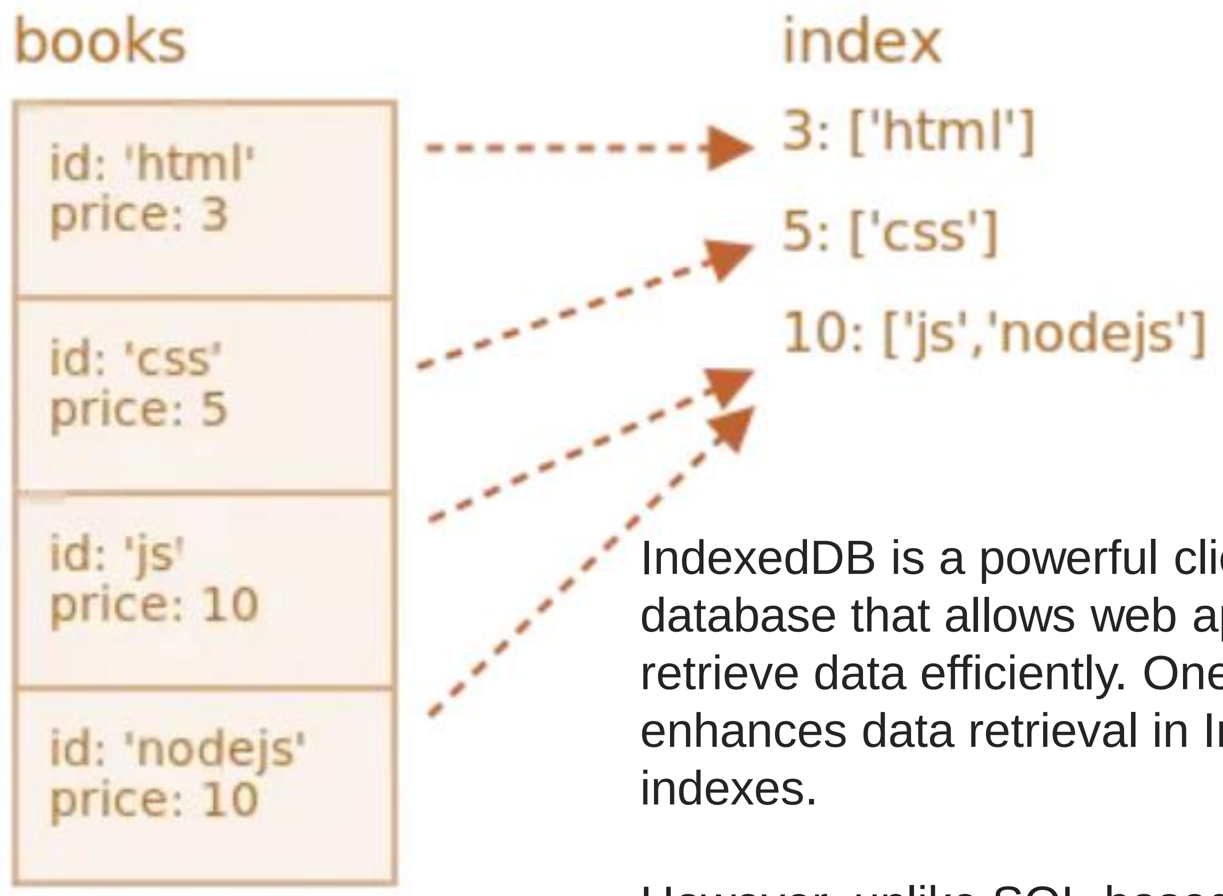
The database is organized around the concept of collections of JSON objects similar to NoSQL databases

IndexedDB is useful for applications that store a large amount of data (for example, a catalog of DVDs in a lending library) and applications that don't need persistent internet connectivity to work (for example, mail clients, to-do lists, and notepads).

Each IndexedDB database is unique to an origin

IndexedDB is built on a transactional database model.

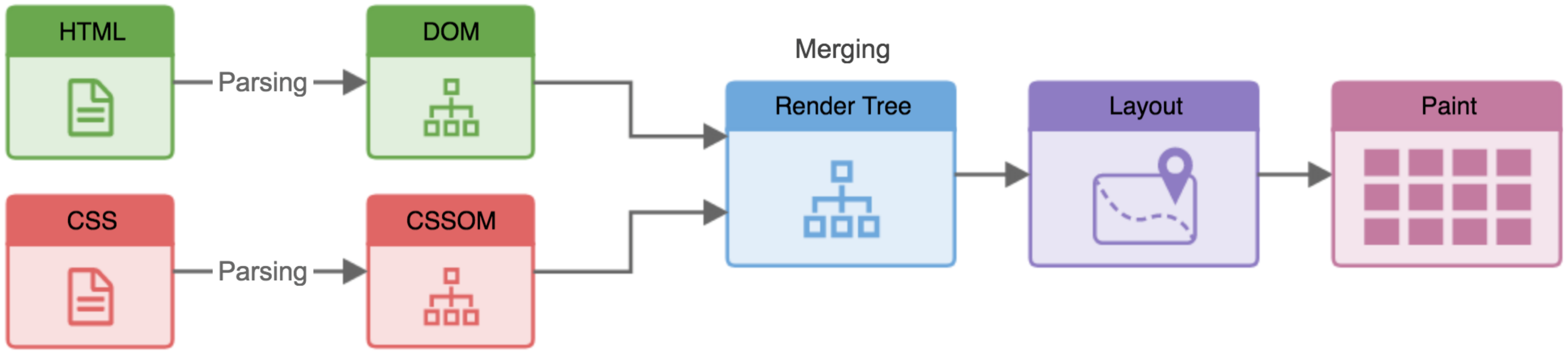
Indexed DB



IndexedDB is a powerful client-side transactional database that allows web applications to store and retrieve data efficiently. One key feature that enhances data retrieval in IndexedDB is the use of indexes.

However, unlike SQL-based RDBMSes, which use fixed-column tables, IndexedDB is a JavaScript-based object-oriented database

Reflow and Repaint



Reflow: compute the layout of each visible element (position and size).
Repaint: renders the pixels to screen.

Repaint Examples of this include *outline*, *visibility*, *background*, or *color*. Repaint is expensive because the browser must verify the visibility of all other nodes in the DOM tree.

Examples that cause reflows include adding or removing content, explicitly or implicitly changing width, height, font-family, font-size and more.

Reflow

Reflow:

The browser recalculates the position and geometry of certain parts of a web page. This can happen after an update on an interactive site, such as changing the width, height, or font-size of a node. Reflow can affect browser performance because it may cause the entire or part of the page layout to be updated.

Common triggers for reflow include changes in:

- Element dimensions (width, height)
- Added/removed elements from the DOM
- Changes in font size or family
- Visibility changes (hidden/shown elements)

Frequent reflows can slow down your web page as the browser juggles recalculating the layout.

Repaint

Repaint:

Repaint is the process where the browser updates the pixels on the screen to reflect the current state of the document. It essentially redraws the portion of the page that has changed due to reflow or other visual modifications.

Repaint is triggered by:

- Changes in element colors or backgrounds
- Applying styles that affect how an element is drawn (e.g., borders, shadows)
- Reflow (as reflow can alter what needs to be drawn)

Similar to reflow, excessive repaints can impact performance.

Event Handling, Reflow, and Repaint

Feature	Description	Triggered By	Impact on Performance
Event Handling	Mechanism for Javascript to respond to user interactions or browser events.	Clicks, scrolls, key presses, window resizing, etc.	Necessary for user interaction, but excessive listeners can slow things down.
Reflow	Process of recalculating the layout of the web page. Determines positions and sizes of elements based on HTML and CSS.	Changes in element dimensions, adding/removing elements, font changes, visibility changes.	Frequent reflows can be expensive as the browser re-layouts the page.
Repaint	Process of updating pixels on the screen to reflect the current visual state.	Changes in element colors, backgrounds, applying styles (borders, shadows), reflow.	Excessive repaints can slow down rendering due to redrawing parts of the page.

Optimizing Reflow and Repaint

Here are some tips to minimize reflows and repaints for a smoother user experience:

•**Minimize DOM manipulation:** Batch DOM updates instead of making many small changes.
•**Use CSS efficiently:** Avoid complex selectors and calculations in your stylesheets.
•**Debounce and Throttle events:** For frequently occurring events, consider techniques like debouncing (delaying execution) or throttling (limiting execution rate) to reduce unnecessary reflows.
•**Use layout properties carefully:** Changing properties like offsetLeft or offsetTop can trigger reflows. Consider alternatives if possible.

BITS Pilani
Pilani Campus

Node.js Ecosystem

Node.js

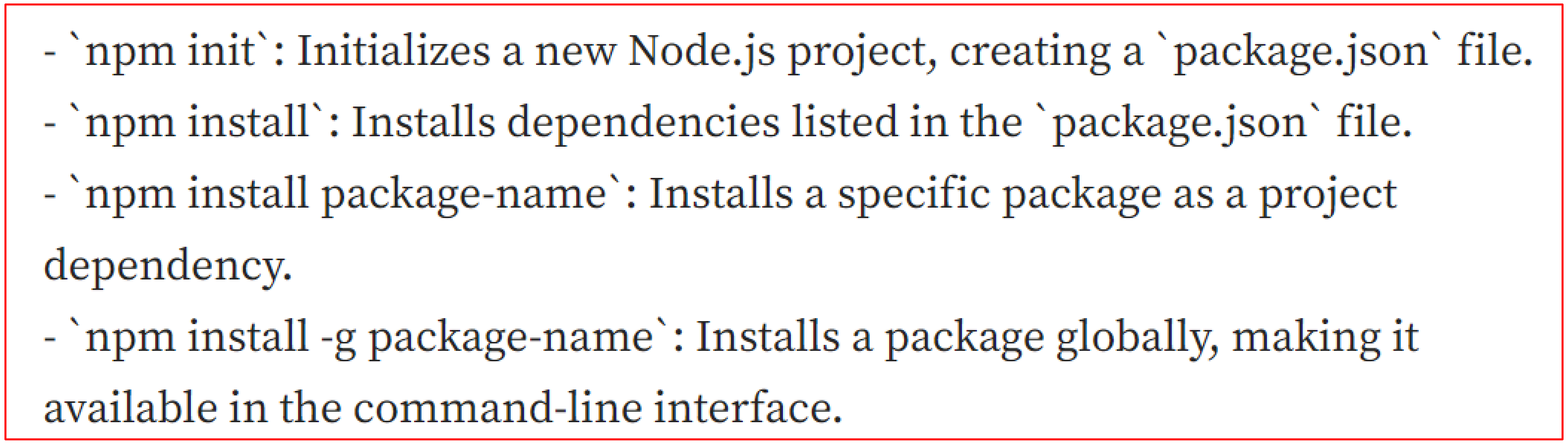
Node.js is an open-source, cross-platform JavaScript runtime environment built on Google Chrome's V8 JavaScript engine.

It allows developers to execute JavaScript code outside of a web browser, making it possible to create server-side applications and command-line tools using JavaScript.

The Node.js ecosystem consists of various libraries, modules, and tools that enhance development productivity and enable developers to build powerful applications.

NPM (Node Package Manager)

- ❑ NPM is the default package manager for Node.js, and it plays a crucial role in the Node.js ecosystem.
- ❑ It allows developers to discover, install, and manage third-party libraries and tools needed for their applications.
- ❑ NPM hosts a vast repository of reusable modules, each serving specific functionalities, which developers can easily integrate into their projects.
- ❑ NPM also enables version management and dependency resolution, ensuring that all dependencies work together seamlessly.



- \`npm init\`: Initializes a new Node.js project, creating a \`package.json\` file.
- \`npm install\`: Installs dependencies listed in the \`package.json\` file.
- \`npm install package-name\`: Installs a specific package as a project dependency.
- \`npm install -g package-name\`: Installs a package globally, making it available in the command-line interface.

Node.js Core Modules

Node.js comes with a set of built-in core modules that provide essential functionalities without the need for additional installations. Some of the core modules include:

- `` `**fs**` ``: File system operations for reading, writing, and manipulating files.
- `` `**http**` ``: Provides HTTP server and client capabilities for building web applications.
- `` `**https**` ``: Similar to `http`, but for secure HTTPS communication.
- `` `**path**` ``: Utilities for working with file paths and directory structures.
- `` `**util**` ``: Utility functions that enhance development productivity.

Express.js

Express.js is a popular, minimalist web framework for Node.js.

It simplifies the process of building web applications by providing essential features and middleware for routing, handling HTTP requests, managing cookies, and more.

Express.js is lightweight and flexible, allowing developers to build APIs, web applications, and even full-fledged websites.

Async Programming

Node.js leverages asynchronous, non-blocking I/O operations, making it highly efficient for handling concurrent connections.

Asynchronous programming is essential for building scalable, high-performance applications that can handle a large number of requests without blocking the event loop.

Node ecoSystem

Nodemon:

Nodemon is a utility tool that monitors changes to your Node.js application’s files and automatically restarts the server when changes are detected. It greatly improves the development workflow, eliminating the need to manually restart the server after every code change.

Babel:

Babel is a powerful **transpiler** that allows developers to write modern JavaScript code using the latest ECMAScript features. It transforms modern JavaScript code into backward-compatible versions, ensuring compatibility with older Node.js versions and browsers.

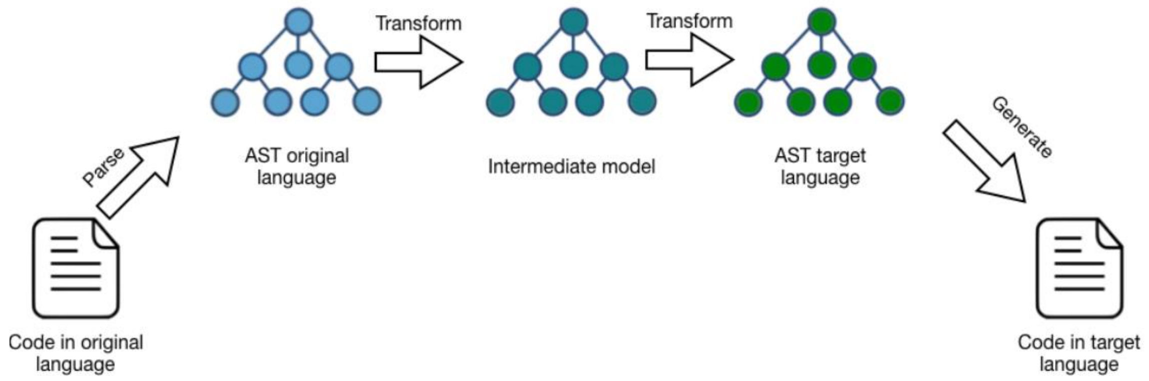
ESLint:

ESLint is a **linting tool** that helps maintain code quality and consistency by enforcing coding standards and best practices. It checks for potential errors, style issues, and code smells, ensuring that the codebase remains clean and easy to maintain.

Transpiler

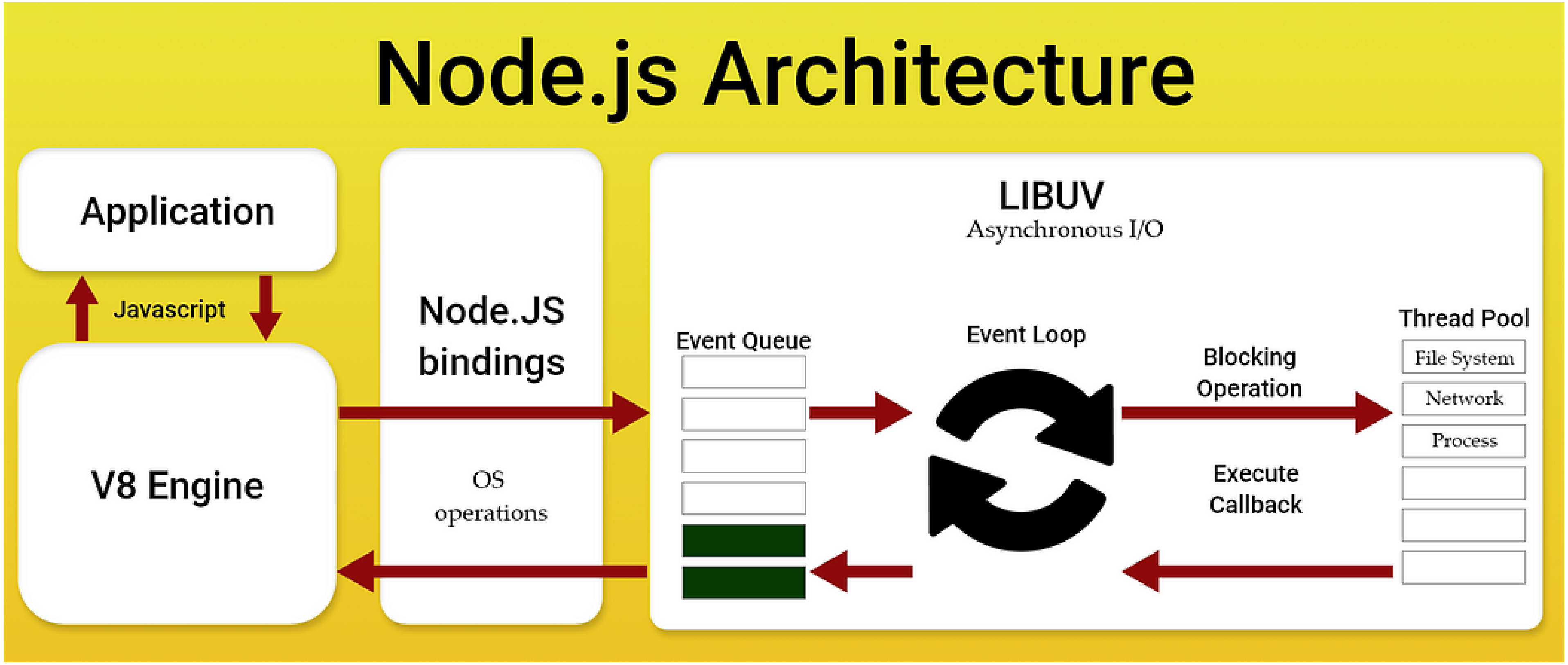
A transpiler, also known as a source-to-source compiler or transcompiler, is a type of translator that takes source code from one programming language and produces an equivalent source code in another language.

- Transpilers perform the following tasks: **syntactic analysis**, **semantic analysis**, and **code generation**
- Transpilers transform code between similar programming languages at the same abstraction level. For example, ES6 to ES5 JavaScript. Transpiler output is meant to be read and edited by developers, hence readability is emphasized.
- Babel is one of the most prominent transpilers. Modern project build systems, such as webpack, provide a means to run a transpiler automatically on every code change.



Transpiler, Compiler and Interpreter

Feature	Transpiler	Compiler	Interpreter
Input	High-level language code	High-level language code	High-level language code
Output	Another high-level code	Machine code or bytecode	No output, executes code
Execution	Requires separate execution	Requires separate execution	Executes code line-by-line
Speed	Faster startup, slower overall	Slower startup, faster overall	Slowest
Error Checking	Basic syntax checks	Full syntax and semantic checks	Line-by-line error checking
Portability	High (target same level)	Lower (targets specific machine)	High (interpreted code)
Examples	Babel (ES6 to ES5)	GCC (C to machine code)	Python interpreter


Node.js Architecture
Application
Javascript
V8 Engine
Node.JS bindings
OS operations
LIBUV
Asynchronous I/O
Event Queue
Event Loop
Blocking Operation
Execute Callback
Thread Pool
File System
Network
Process

Architecture (cont.)

Event Loop: The Event Loop is the heart of Node.js architecture. It allows Node.js to handle asynchronous operations efficiently. When a Node.js application starts, it initializes the Event Loop, which continuously listens for events and executes their associated callback functions. The Event Loop is single-threaded, meaning it runs on a single main thread, but it can process multiple I/O operations concurrently.

Event Emitters and Listeners: In Node.js, many objects are capable of emitting events. These objects are known as Event Emitters. Examples of event emitters include HTTP servers, file system operations, and database connections. Other parts of the application can register event listeners to handle specific events emitted by these objects. When an event occurs, the corresponding listener’s callback function is executed asynchronously.

Architecture (cont.)

Callbacks and Non-Blocking I/O: Node.js relies heavily on callbacks, which are functions passed as arguments to asynchronous functions. When an asynchronous operation completes (e.g., reading a file, making an HTTP request), the associated callback function is executed, allowing the application to continue processing other tasks without waiting for the I/O operation to complete. This non-blocking I/O model enables Node.js to handle multiple concurrent requests efficiently.

Libuv: Libuv is a C library that provides the platform-specific implementation of the Event Loop and handles I/O operations for Node.js. It abstracts the underlying operating system's asynchronous I/O capabilities, allowing Node.js to be cross-platform. Libuv handles tasks such as file system operations, timers, and networking.

Architecture (cont.)

Modules: Node.js follows a modular architecture, where functionality is encapsulated into reusable modules. Modules can be custom-built or obtained from the Node Package Manager (NPM). The modular approach promotes code reusability and maintainability, as each module is responsible for specific functionality.

V8 Engine: Node.js uses the V8 JavaScript engine, which is developed by Google and also used in the Chrome browser. V8 compiles JavaScript code to native machine code, making it highly performant. This engine enables Node.js to execute JavaScript code efficiently outside of a browser environment.

HTTP and HTTPS Modules: Node.js includes built-in modules for creating HTTP and HTTPS servers, making it easy to build web applications and APIs. These modules allow developers to handle incoming HTTP requests, process responses, and manage server-side routing.

Remarkable Features

- **Asynchronous and Non-blocking I/O**: Node.js uses an event-driven, non-blocking I/O model, which allows it to handle multiple concurrent connections efficiently. Asynchronous operations ensure that the application doesn’t get blocked while waiting for I/O operations to complete, making it highly scalable and suitable for real-time applications.
- **Single-Threaded Event Loop**: Node.js employs a single-threaded event loop that manages all I/O operations and callbacks. This simplifies concurrency management and avoids the complexity of handling multiple threads, making development easier.
- **NPM** (Node Package Manager): NPM is a vast repository of reusable libraries and modules that developers can easily integrate into their projects. It facilitates code sharing and drastically speeds up the development process by providing access to thousands of open-source packages.
- **Cross-platform**: Node.js applications can run on various platforms, including Windows, macOS, and Linux, without any modifications. This cross-platform support ensures that developers can deploy their applications on different operating systems seamlessly.

Remarkable Features

- **Fast Execution**: Node.js is built on the V8 JavaScript engine, developed by Google, which compiles JavaScript code to native machine code. This results in high performance and quick execution of Node.js applications.
- **Scalability**: Node.js is highly scalable due to its non-blocking I/O model and lightweight architecture. It can handle a large number of concurrent connections with minimal resource usage, making it suitable for building high-performance, scalable applications.
- **Event-driven Architecture**: Node.js uses event emitters and listeners to manage events and callbacks, enabling efficient handling of asynchronous tasks. This event-driven architecture simplifies handling I/O operations and promoting modular and clean code.

Applications on Node.js

Web Applications

- Node.js is commonly used for building web applications and websites. Its asynchronous and event-driven nature allows it to handle a large number of concurrent connections efficiently, making it ideal for real-time applications and websites with high traffic

API Development

- Node.js is an excellent choice for creating APIs to expose functionalities and data to other applications. Its lightweight and fast nature makes it well-suited for building API endpoints that respond quickly to incoming requests.

Applications on Node.js

Real-Time Applications

- Node.js is a popular choice for building real-time applications, such as chat applications, online gaming servers, collaboration tools, and social media platforms. Its event-driven architecture and support for WebSockets enable seamless real-time communication between clients and servers.

Microservices

- Node.js is well-suited for building microservices architectures, where applications are broken down into smaller, independent services. Each microservice can be developed and deployed separately, and Node.js' lightweight nature makes it a good fit for these individual components.

Applications on Node.js

Streaming Applications

- Node.js is capable of handling streaming data efficiently, making it suitable for applications dealing with real-time data processing, video and audio streaming, and file uploads and downloads.

Single Page Applications (SPAs)

- Node.js is often used in conjunction with front-end JavaScript frameworks like React, Angular, or Vue.js to build single-page applications (SPAs). The combination of Node.js on the server-side and a JavaScript framework on the client-side enables seamless communication between the client and server.

innovate achieve lead

BITS Pilani

Pilani Campus

React Overview

Overview https://react.dev/learn

React is a popular JavaScript library for building user interfaces (UIs) efficiently. It was created by Facebook and is now widely used for developing single-page web applications (SPAs) and other interactive web experiences.

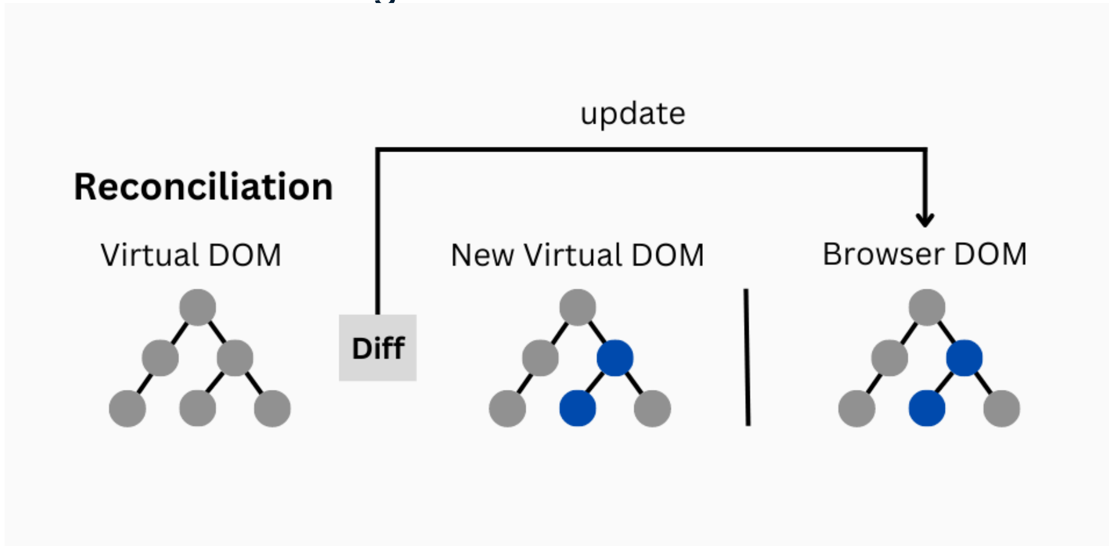
Key Concepts of React

- **Component-Based Architecture:** React applications are built by composing reusable UI components. Each component manages its own state and renders its UI based on that state. This promotes code modularity and maintainability.
- **Declarative Style:** React components describe what the UI should look like for a given state. React then efficiently determines how to update the DOM (Document Object Model) when the state changes. This makes reasoning about your UI and debugging easier.
- **Virtual DOM:** React employs a virtual DOM, a lightweight representation of the real DOM. When the state of a component changes, React updates the virtual DOM first. Then, it calculates the most efficient way to update the actual DOM, minimizing unnecessary manipulations. This significantly improves performance.
- **JSX (JavaScript XML):** JSX is a syntax extension for JavaScript that allows you to write HTML-like structures within your code. This makes it easier to visualize and write UI components. However, JSX is transformed into regular JavaScript before the browser executes it.

Virtual DOM

The Virtual DOM is a lightweight JavaScript representation of the DOM. When a component's state changes, React updates the Virtual DOM and compares it to the previous version. React then updates the actual DOM to match the changes in the Virtual DOM. This process is called reconciliation.

The Virtual DOM is one of the reasons why React is so fast. By only updating the parts of the DOM that have changed, React can avoid the overhead of re-rendering the entire DOM.



JSX

JSX, which stands for JavaScript XML, is a syntax extension for JavaScript that allows you to write HTML-like structures within your code. It's primarily used with React but can also be used with other libraries.

What it Looks Like:

JSX resembles HTML, but it's actually JavaScript code.

```
function Header()
 {
   return ( <h1>Welcome to WILP FSAD!</h1> );
}
```

In this example, the Header function returns a JSX element representing an <h1> tag with the text "Welcome to …!".

JSX

Benefits of JSX:

- **Readability:** JSX makes it easier to write and understand UI components because the code resembles the structure of the UI itself.
- **Maintainability:** Separating UI logic from JavaScript code can improve code organization and maintainability.
- **Declarative Style:** JSX encourages a declarative style of programming, where you describe what the UI should look like for a given state.

Not Required: While popular with React, JSX is not mandatory. You can write React applications without it, but it's widely adopted for its readability benefits.

Syntactic Restrictions: JSX has some limitations compared to HTML. For example, you can't use self-closing tags like <br /> directly in JSX.

Props

Props in React are arguments passed to React components. Props are immutable, meaning they cannot be changed inside the component. Props are read-only.

To pass props to a component, you can use the following syntax:

```
<MyComponent prop1="value1" prop2="value2" />
```

To access props inside a component, you can use the this.props object. For example:

```
class MyComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.prop1}</h1>
        <h2>{this.props.prop2}</h2>
      </div>
    );
  }
}
```

State

In React, state is a JavaScript object that stores data that can be used to influence the rendering of a component. When a component's state changes, the component re-renders with the updated data.

There are two ways to manage state in React: using the **useState** hook and using class components.

Using the useState hook

The useState hook is a function that returns a pair of values: the current state value and a function that updates the state value. To use the useState hook, you first need to import it from the React library:

```
import React, { useState } from 'react’;
```

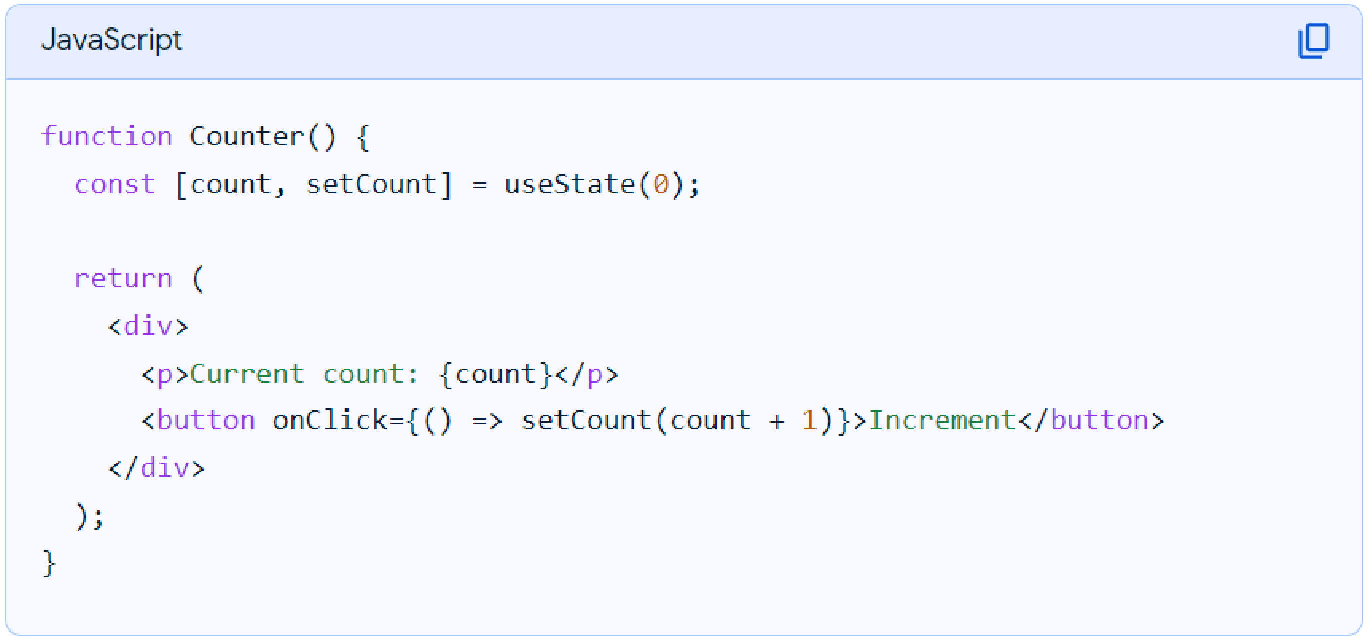
Then, you can use the useState hook to declare a state variable:

```
const [count, setCount] = useState(0);
```

State (examples)

The count variable is the current state value, and the setCount function is the function that updates the state value.

You can use the count variable in your component's render() method:



JavaScript

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

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

Conditional Rendering

```
function Message(props) {
  if (props.isLoggedIn) {
    return <p>Welcome back, {props.username}!</p>;
  } else {
    return <p>Please log in to see your messages.</p>;
  }
}
```

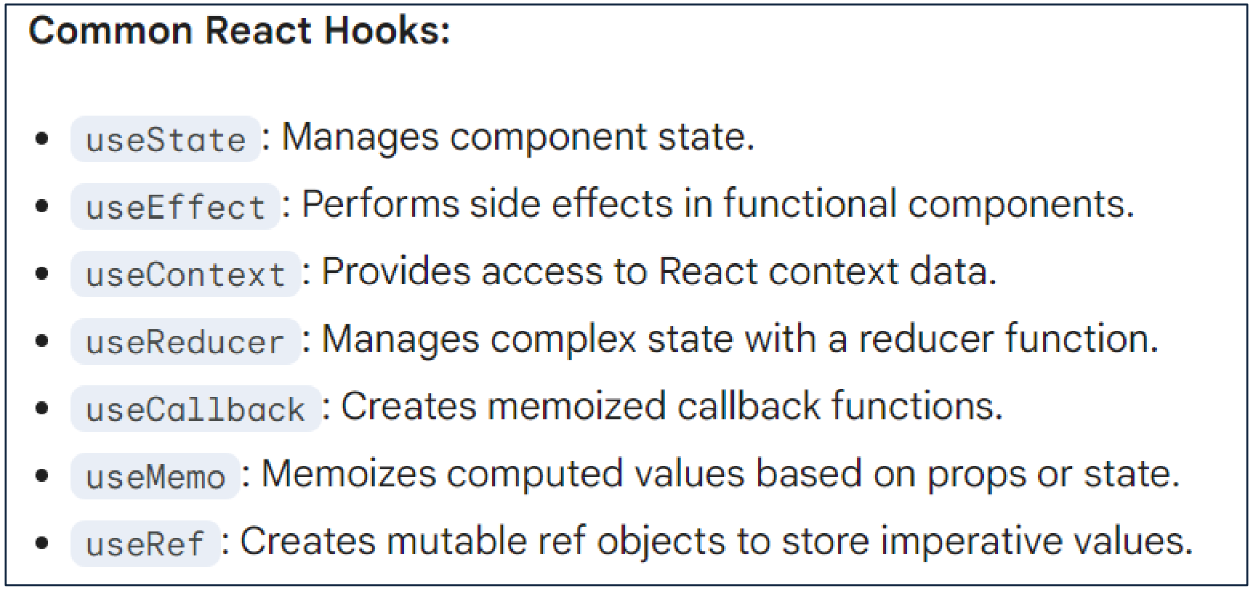
```
function ShowContent(props) {
  if (props.isLoading) {
    return <p>Loading...</p>;
  }

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

```
function ShowContent(props) {
  return (
    <div>
      {props.isLoading && <p>Loading...</p>}
      {!props.isLoading && (
        <>
          <h1>{props.title}</h1>
          <p>{props.content}</p>
        </>
      )}
    </div>
  );
}
```

React Hooks

React hooks are a powerful addition to React introduced in version 16.8. They allow you to "hook into" state and other React features from functional components. This enables functional components to have more functionality previously only available with class components.



Common React Hooks:

- `useState`: Manages component state.
- `useEffect`: Performs side effects in functional components.
- `useContext`: Provides access to React context data.
- `useReducer`: Manages complex state with a reducer function.
- `useCallback`: Creates memoized callback functions.
- `useMemo`: Memoizes computed values based on props or state.
- `useRef`: Creates mutable ref objects to store imperative values.

React Hooks

Functional Component Advantages: Hooks let you leverage the simplicity and clarity of functional components while still managing state and side effects.

Improved Code Readability: Hooks often lead to cleaner and more concise code compared to class components with lifecycle methods.

Easier Testing: Functional components with hooks are generally easier to test and reason about due to their simpler structure.

React Routing

React Router is a popular library for implementing client-side routing in React applications. It enables you to navigate between different UI components (views) based on URL changes without reloading the entire page.

This provides a smoother and more responsive user experience for single-page applications (SPAs).

Route

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

// Define your components
const Home = () => <h1>Home Page</h1>;
const About = () => <h1>About Us</h1>;
const Contact = () => <h1>Contact Us</h1>;

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />  {/* Route for home page */}
        <Route path="/about" element={<About />} />  {/* Route for about page */}
        <Route path="/contact" element={<Contact />} />  {/* Route for contact page */}
      </Routes>
    </Router>
  );
};

export default App;
```

Advantages

Improved User Experience: Client-side routing allows for smoother transitions between views without full page reloads.

Deep Linking: Users can bookmark specific views within your application using URLs.

Component Reusability: Routes can be associated with reusable components, promoting code organization.

Flexibility: React Router offers features for handling nested routes, dynamic parameters, and route guards for authorization.

Redux
https://react-redux.js.org/tutorials/quick-start

Redux is a popular library for managing application state in JavaScript applications, particularly those built with React. It provides a predictable way to store and update your application's data in a central location, making it easier to reason about how your UI responds to changes.

Redux Concepts:

Store: A central location that holds the entire state of your application. It provides a ***dispatch*** method to update the state.

Actions: Plain JavaScript objects that describe what happened in the application. They include a *type* property and an optional *payload* containing data.

Reducers: Pure functions that take the current state and an action as arguments, and return a new state object based on the action type and payload.

Why to use?

Centralized State Management: As your React application grows, managing application state through prop drilling (passing data down through component hierarchies) becomes cumbersome and error-prone. Redux provides a single source of truth for your application's state, making it easier to access and update data from anywhere in your components.

Predictable State Updates: Redux enforces a unidirectional data flow. Changes to the application state are made through actions, which are then processed by pure reducer functions. This predictability makes it easier to debug and reason about how your application behaves as the state changes.

Scalability: For complex applications with a lot of data flow, Redux provides a structured approach to managing state. This becomes especially beneficial as your application grows in size and complexity.

Improved Developer Experience: Redux offers features like time-travel debugging (stepping back through past states) and a thriving community with many helpful libraries and tools. These features can streamline development and make it easier to maintain your codebase.

But

However, Redux might not be for every project. Here are some things to consider:

- **Complexity:** Setting up and managing Redux can add complexity to smaller or simpler React applications. If your application's state management needs are relatively straightforward, Redux might be overkill.
- **Learning Curve:** There's a learning curve associated with understanding Redux concepts and patterns. If you're new to React or state management, it might be beneficial to explore simpler solutions first.

Alternatives to Redux:

- React Context API: A built-in React feature for sharing data across components without prop drilling. It's a good option for simpler applications or managing data shared by a limited set of components.
- Zustand: A lightweight state management library with a simpler API compared to Redux. It might be a good choice for projects that need basic state management without the overhead of Redux.

Review Questions

1. Given a scenario where you have to design a highly scalable Node.js application for real-time chat messaging. Discuss the architecture considerations, potential bottlenecks, and strategies to optimize performance and ensure the reliability of the chat system under heavy loads.
2. In a microservices architecture, Node.js is often used to develop lightweight, scalable backend services. Discuss the advantages and challenges of using Node.js for microservices development. Explain how you would implement service discovery, load balancing, fault tolerance, and distributed tracing to ensure reliability and scalability in a Node.js-based microservices ecosystem.
3. Consider a scenario where you're building a complex user interface component in React.js that involves handling dynamic state changes, data fetching, and performance optimizations. Describe how you would leverage React hooks, context API, and memorization techniques to manage state effectively, optimize re-renders, and ensure smooth user interactions.
4. Imagine you're tasked with optimizing the rendering performance of a large React application. Describe common techniques and best practices you would employ to minimize initial load times, reduce time to interactive, and optimize rendering performance for both client-side and server-side rendering scenarios.