Module-3

Form enhancement and validation. Introduction to

MERN: MERN components, Server less Hello world.

React Components: Issue Tracker, React Classes, Composing Components, Passing Data Using Properties, Passing Data Using Children, Dynamic Composition.

MERN

INTRODUCTION TO MERN AND MERN COMPONENTS

The MERN stack is a widely adopted full-stack development framework that simplifies the creation of modern web applications. Using JavaScript for both the frontend and backend enables developers to efficiently build robust, **scalable, and dynamic applications.**

How MERN Works Together?

- Frontend (React.js) sends a request to the backend.
- Backend (Express.js + Node.js) processes the request.
- Database (MongoDB) stores/retrieves the requested data.
- Backend sends the data back to the frontend for display.

Why Use MERN Stack?

- Full-stack JavaScript-based framework.
- Fast development with reusable components.
- Scalable and handles real-time applications.
- Supported by a large developer community.

Mern Components

React (Frontend Component)

- React is an open-source JavaScript library developed by Facebook.
- Used for building user interfaces (UI) rendered in HTML.
- Unlike AngularJS, React is a library, not a full framework.
- Handles the "View" (V) part of MVC architecture.

Real-world Usage:

- Used by companies like Facebook, Airbnb, Atlassian, Bitbucket, Disqus, Walmart.
- Highly popular – over 120,000 stars on GitHub.

Why Facebook Invented React

- Originally built for Facebook Ads team.
- Traditional MVC with two-way data binding led to complex, hard-to-maintain code.
- Frequent cascading updates made apps error-prone.
- Solution: Declarative UI – re-render view when data changes, instead of manipulating DOM manually.

Features of React

1. Declarative Views

- React handles UI updates automatically.
- Programmer defines how UI should look for a given state.
- When data changes, React uses Virtual DOM to determine minimal updates to real DOM.

2. Virtual DOM

- A lightweight copy of actual DOM stored in memory.
- React compares new Virtual DOM with old Virtual DOM → updates only what has changed.
- Ensures high performance with minimal DOM manipulation.

3. Component-Based Architecture

- Everything in React is a component:
    - Components are independent, reusable, and maintain their own state.
    - Can be combined together to build complex UIs.
- **Parent-child communication via:**
    - Props (read-only) – from parent to child.
    - Callbacks – from child to parent.

4. No Templates

- React doesn't use a separate template language.
- Uses JavaScript and JSX to build UI.

- JSX = JavaScript XML → similar to HTML but written in JS files.
- JSX simplifies creation of nested HTML elements.

5. Isomorphic Code

- React code can run on both browser and server.
- Useful for Server-Side Rendering (SSR) → better SEO and faster initial load.

Node.js

- JavaScript runtime environment (outside the browser).
- Built on Chrome's V8 engine.
- Can run full applications in JavaScript (like backend APIs).
- Used by companies like Netflix, Uber, LinkedIn.

Node.js Modules

- Browser JS needs HTML to include multiple files, Node.js doesn't.
- Node.js uses CommonJS module system:
  - Split code across files/modules for better organization.
  - Use require() to include modules.
- Comes with built-in core modules (File System, Network, etc.).
- Supports third-party modules via npm.

npm (Node Package Manager)

- Default package manager for Node.js.
- Can install:
    - Third-party libraries like React, jQuery.
    - Dependencies for your project.
- Provides:
    - Huge collection of reusable packages.
    - Version conflict resolution.
- Tools like Webpack or Browserify bundle all modules for browser use.
- npm is now the largest package ecosystem, even surpassing Maven.

Node.js – Event Driven Architecture

- Uses asynchronous, non-blocking I/O.
- Doesn't rely on threads – uses callbacks and event loop.
- Example:
    - Instead of waiting for a file to open, you pass a callback and move on.
- Event loop keeps checking for events and executes their callbacks.
- Similar to AJAX style asynchronous code.
- Efficient, but requires learning asynchronous coding patterns.

Express.js – Web Server Framework for Node.js

What is Express?

- A **web framework** for Node.js – simplifies writing server code.
- Without Express, writing a full web server using pure Node.js is **tedious**.
- Provides an easy way to define:
    - **Routes** (URL patterns).
    - **Request handling logic**.

Routing

- Routes in Express are defined using **regex-like patterns**.
- Each route has a **handler function**:
    - Takes in the parsed **HTTP request**.
    - Sends a response based on business logic.

Request/Response Handling

- Express parses:
    - URL
    - Headers
    - Parameters
- Helps you easily:
    - Set response codes
    - Set cookies
    - Send custom headers

Middleware

- Middleware = **custom reusable functions** used in request/response flow.
- Use cases: **logging**, **authentication**, **error handling**, etc.

Template Engine Support

- Express doesn't have a built-in template engine.
- Supports third-party engines like:
    - pug
    - mustache
- **Note**: For SPAs (Single Page Applications), template engines are usually unnecessary as the dynamic rendering happens on the **client-side** using React.

MongoDB – NoSQL Database for MERN Stack

What is MongoDB?

- A **NoSQL**, **document-oriented** database.
- Used for storing data in **JSON-like documents**.
- Schema is **flexible**, making it ideal for rapidly evolving applications.

Who uses it?

- Big companies: **Facebook**, **Google**, **SAP**, **Royal Bank of Scotland**.

NoSQL Basics

- **NoSQL** = *non-relational* (no tables, rows, strict relations).
- Key benefits:
    1. **Horizontal scalability** (distributes load across servers).
        - Sacrifices strong consistency (refer **CAP theorem**).
    2. **Object-document mapping** aligns with how data is used in applications.

Object vs Relational Mismatch

- Traditional RDBMS requires mapping objects to rows/columns.
- This mismatch requires **ORMs** (e.g., Hibernate).
- MongoDB stores data **as-is** (in object/document format), removing the need for complex mappings.

Document-Oriented Storage

- Data stored as **documents** (JSON-like).
- Documents grouped into **collections** (similar to tables).
- Each document has a unique **identifier**, automatically indexed.

Example:

Invoice with multiple items

- In RDBMS:
    - Two tables: invoice, invoice_items (linked via foreign key).
- In MongoDB:

- One document with all invoice details and item list as nested fields.

Downsides

- Data is **de-normalized** (can be duplicated).
- Storage usage increases.
- Operations like renaming a master entry require updating multiple documents.
- But storage is cheap, and such updates are rare.

Schema-Less

- No fixed schema required for documents.
- Fields can vary between documents in the same collection.
- Quick development without database migrations.
- **Downside**: Data consistency must be ensured **in code**.
- Solution: Use ODMs like **Mongoose** (adds structure and validation).

JavaScript-Based Query Language

- Uses **JSON syntax** instead of SQL.
- Queries are built using **JavaScript-style JSON objects**.
- Easier for developers working with JS (like in MERN).

JSON vs BSON

- Internally, MongoDB uses **BSON** (Binary JSON) for efficiency.
- Communication and operations use JSON format.

Mongo Shell

- Comes with a shell interface powered by JavaScript (like Node.js).
- You can:
    - Interact with DB using JS.
    - Write **stored procedures** (JS snippets run on server).

Tools and Libraries in MERN Stack Development

◈ React-Router

React handles rendering and user interactions in components.

Routing enables transition between views and syncs URLs.

React-Router:

- ➢ Parses URLs and maps them to components.
- ➢ Handles browser history (e.g., Back button).
- ➢ Mimics page transitions without full reloads.
- ➢ React-Router simplifies navigation in SPAs.

◈ React-Bootstrap

React adaptation of Bootstrap, a popular CSS framework.

Provides pre-built, customizable UI components.

Useful for:

Fast UI development.

Learning to design custom components.

Alternative UI libraries:

Material-UI (MUI)

Elemental UI

◈ Webpack

Bundles and modularizes client-side code.

Converts JSX to JavaScript (compilation step).

Competing tools: Bower, Browserify, Gulp, Grunt

Simplifies delivery of client-side code to browsers.

◈ Other Libraries

Server-side:

body-parser: Parses JSON/form data in POST requests.

ESLint: Ensures code quality and consistency.

Client-side:

react-select and other useful component libraries.

◈ Other Popular Libraries

Redux: State management for complex apps.

Mongoose: ODM (Object Document Mapper) for MongoDB.

Jest: Testing library for React apps.

Server Less Hello World

Introduction

- **What is Serverless?**
  Serverless computing allows developers to run applications without managing servers. Cloud providers handle infrastructure, scaling, and maintenance.
- **Why Serverless?**
  - No need to manage servers
  - Auto-scaling and cost-effective

    - Faster deployment
    - Ideal for event-driven applications
- **Popular Serverless Providers**
    - AWS Lambda
    - Azure Functions
    - Google Cloud Functions
    - Netlify Functions
    - Vercel Serverless

Objective:

To create a minimal React app without using Node.js, Express, or any installations—just a browser and a single HTML file.

Tools Used:

React: JavaScript library for building UI.

ReactDOM: Converts React components to actual DOM elements.

CDN: Libraries are loaded via CDN using <script> tags.

React: https://unpkg.com/react@16/umd/react.development.js

ReactDOM: https://unpkg.com/react-dom@16/umd/react-dom.development.js

Step-by-Step Explanation:

Create an HTML file

Save it as index.html.

Add React and ReactDOM libraries

Place these in the <head> section:

```
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
```

Create a target in <body>

This is where your React component will render.

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

Create a React element using React.createElement()

```
const element = React.createElement('div', {title: 'Outer div'},
  React.createElement('h1', null, 'Hello World!')
);
```

This creates a nested element:

Outer with a title attribute.

Inner <h1> element with text "Hello World!".

Render it using ReactDOM.render()

ReactDOM.render(element, document.getElementById('content'));

Complete Code (index.html)

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
  <title>Pro MERN Stack</title>
  <script
src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"></script>
</head>
<body>
  <div id="content"></div>
  <script>
    const element = React.createElement('div', {title: 'Outer div'},
      React.createElement('h1', null, 'Hello World!')
```

```
    );
    ReactDOM.render(element, document.getElementById('content'));
  </script>
</body>
</html>
```

- No installations required: runs directly in browser.
- Uses CDN links to load React libraries.
- React.createElement() is used instead of JSX (we'll explore JSX later).
- React elements are JavaScript objects (virtual DOM).
- ReactDOM.render() injects the virtual DOM into the real DOM.

Output:

Displays "Hello World!" on the browser.

Hovering over the text shows a tooltip "Outer div".

Creating the Express Project

1. Installing Node.js Using nvm

What is Node.js?

Node.js lets us run JavaScript outside a browser (on the server).

Why use nvm (Node Version Manager)?

Helps us install and switch between multiple Node.js versions.

Useful when projects need different Node versions.

Steps:

1. Open Terminal (Command Line).

Install nvm

```
nvm install --lts
```

Check installed version:

```
node -v
```

Now Node.js and npm are ready to use!

2. Initializing a Project with npm

What is npm?

- Stands for Node Package Manager.
- Manages packages (libraries) for Node.js.

To start a new project:

Create a project folder:

```
mkdir issuetracker

cd issuetracker
```

Initialize with npm:

```
npm init -y
```

◈ Creates a package.json file with default values.

3. Installing Express Framework

What is Express?

- A web framework for Node.js.
- Helps create server-side applications easily.

To install Express:

```
npm install express
```

This adds Express as a dependency in package.json.

4. Understanding package.json

What is package.json?

- A file that stores info about the project and dependencies.

Key Fields:

name: Project name.

version: Project version.

scripts: Commands you can run (like npm start).

dependencies: List of installed packages (like Express).

5. Serving Static Files Using Express

Static files = HTML, CSS, JS, images

We serve them to the client (browser).

Steps in code:

Create a folder public/ and put your static files there.

In your server.js file:

```
const express = require('express');
const app = express();
app.use(express.static('public'));
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Now, if public/index.html exists, open in browser:

Output

http://localhost:3000/ (follow this link)

Summary

Concept	Purpose
nvm	Manage Node.js versions
npm init	Start a new Node.js project
express	Framework to build servers
package.json	Project config and dependencies
express.static()	Serve HTML/CSS/JS files

React Components

🔷 Why Use React Components?

- JSX alone is limited—real-world apps need interactivity, data handling, and reusability.
- Components:
    - Can be composed of other components and HTML elements.
    - React to user input, manage state, and interact with each other.
    - Make the app modular, maintainable, and scalable.

Application: Issue Tracker

This app mimics basic GitHub Issues/Jira functionality.

◆ **Features:**

- View a list of issues (with filters).
- Add new issues.
- Edit/update issues.
- Delete issues.

◆ **Attributes of an Issue:**

- title: summary (text)
- owner: assigned user (text)
- status: open/assigned/resolved etc. (list)
- created: auto-set date
- effort: estimated days (number)
- due: optional due date (date)

React Classes

Purpose:

To move from simple JSX to structured, reusable React components.

syntax:

```
class HelloWorld extends React.Component {
  render() {
    const continents = ['Africa','America','Asia','Australia','Europe'];
    const helloContinents = Array.from(continents, c => `Hello ${c}!`);
    const message = helloContinents.join(' ');
    return (
      <div title="Outer div">
        <h1>{message}</h1>
      </div>
    );
  }
}
```

Instantiate a component:

```
const element = <HelloWorld />;
ReactDOM.render(element, document.getElementById('contents'));
```

Important Notes:

- **React class components** extend React.Component.
- The render() method is **mandatory** and must return an element.
- JSX must return **a single root element**.
- <HelloWorld /> is an instance of the component, like .

- Use **es2015 classes** with class and extends.

Creating a React Class Component

- A **class component** is created by extending React.Component.
- It must include a render() method, which returns JSX (React's syntax similar to HTML).

Example

```
class HelloWorld extends React.Component {
  render() {
    return <h1>Hello, World!</h1>;
  }
}
```

Understanding the render() Method

- The render() method **must** be present in a class component.
- It returns JSX, which defines what should be displayed in the UI.
- Example of render() with data processing inside:

```
render() {
  const continents = ['Africa', 'America', 'Asia', 'Australia', 'Europe'];
  const helloContinents = continents.map(c => `Hello ${c}!`);
  const message = helloContinents.join(' ');
```

```
  return <h1>{message}</h1>;
}
```

Why Use React Classes?

- They allow more **structured and reusable** components.
- They support **lifecycle methods** (to manage component behavior).
- They help in handling **state and complex UI logic**.

Creating a React Component

- You can define a React class component using class ComponentName extends React.Component {}.
- Example:

```
class HelloWorld extends React.Component {
  render() {
    return <h1>Hello, World!</h1>;
  }
}
```

- This can be used like <HelloWorld /> in JSX.

Rendering a Component

- Use ReactDOM.render() to display the component in the DOM.
- Example:

```
const element = <HelloWorld />;

ReactDOM.render(element, document.getElementById('contents'));
```

Component Composition in React

Component composition is a concept in React where **large UI elements** are split into **smaller, reusable components**. Instead of building a **monolithic** UI (everything in one big file), we **break it down into independent parts**.

For example, imagine a page where users can:

✓ **Filter issues**

✓ **See a table of issues**

✓ **Add a new issue**

Instead of writing all this logic in one component, we create three smaller **components**:

- IssueFilter → Handles filtering issues
- IssueTable → Displays a list of issues
- IssueAdd → Allows users to add a new issue

This makes the code **modular**, **readable**, and **easy to manage**.

Step-by-Step Implementation

1. Creating Placeholder Components

Open App.jsx and **define three components**:

```
import React from "react";
class IssueFilter extends React.Component {
  render() {
    return <div>This is a placeholder for the issue filter.</div>;
  }
}
class IssueTable extends React.Component {
  render() {
    return <div>This is a placeholder for the issue table.</div>;
  }
}
class IssueAdd extends React.Component {
  render() {
    return <div>This is a placeholder for adding an issue.</div>;
  }
}
export { IssueFilter, IssueTable, IssueAdd };
```

Composing the Components into One Parent (IssueList)

Now, create another component **IssueList** that groups these three:

```
import React from "react";
import { IssueFilter, IssueTable, IssueAdd } from "./App";
class IssueList extends React.Component {
  render() {
    return (
      <>
        <h1>Issue Tracker</h1>
        <IssueFilter />
        <hr />
        <IssueTable />
        <hr />
        <IssueAdd />
      </>
    );
  }
}

export default IssueList;
```

Rendering IssueList in main.jsx

Modify main.jsx to display the IssueList component:

```
import React from "react";
import ReactDOM from "react-dom/client";
import IssueList from "./App";

ReactDOM.createRoot(document.getElementById("root")).render(
  <IssueList />
);
```

What Happens Here?

issueList is the **main parent component**

It **composes** IssueFilter, IssueTable, and IssueAdd inside it

Everything is displayed in the browser in a **structured and reusable way**

```
myexpressapp/
|-- backend/              # Your Express.js backend (if any)
|-- frontend/             # Your React frontend
|   |-- src/              # React source files
|   |   |-- components/   # Create this folder for reusable components
|   |   |   |-- IssueFilter.jsx
|   |   |   |-- IssueTable.jsx
|   |   |   |-- IssueAdd.jsx
|   |   |   |-- IssueList.jsx  <-- Create this file here!
|   |   |-- App.jsx
|   |   |-- main.jsx
|-- package.json
```

Passing Data Using Properties

Objective:

To make components reusable by passing different data (props) from parent to child components.

Key Concept: Props

- Props (short for properties) allow you to pass data from a parent component to a child component.
- In JSX, you pass props just like HTML attributes.
- In the child component, they're accessed using this.props.

◈ Example Use Case: IssueRow Component

We want to render each issue (from the issue tracker) as a row in a table using a reusable IssueRow component.

IssueTable Structure:

```
class IssueTable extends React.Component {
  render() {
    const rowStyle = { border: "1px solid silver", padding: 4 };
    return (
      <table style={{ borderCollapse: "collapse" }}>
        <thead>
          <tr>
```

```
            <th style={rowStyle}>ID</th>
            <th style={rowStyle}>Title</th>
          </tr>
        </thead>
        <tbody>
          <IssueRow
            rowStyle={rowStyle}
            issue_id={1}
            issue_title="Error in console when clicking Add"
          />
          <IssueRow
            rowStyle={rowStyle}
            issue_id={2}
            issue_title="Missing bottom border on panel"
          />
        </tbody>
      </table>
    );
  }
}
```

IssueRow Component:

```
class IssueRow extends React.Component {
  render() {
    const style = this.props.rowStyle;
    return (
      <tr>
        <td style={style}>{this.props.issue_id}</td>
        <td style={style}>{this.props.issue_title}</td>
      </tr>
    );
  }
}
```

Notes on JSX:

JSX does not support HTML-style comments (<!-- -->). Use JavaScript-style comments inside {}:

{/* This is a JSX comment */}

Props can be strings, numbers, objects, etc.

Use {} for JS expressions (numbers, objects).

Use "" for strings.

React Style Prop

React requires style to be an object, not a string:

```
const rowStyle = { border: "1px solid silver", padding: 4 };
```

Inline style needs double braces:

```
<table style={{ borderCollapse: "collapse" }}>
```

Passing Data Using children in React

Concept Overview

In addition to using props (custom attributes) to pass data to components, React also allows passing data using the component's children, much like nesting elements in HTML.

This allows more flexibility, especially when the data you pass is:

- Rich content (like JSX)
- Multiple nested components
- Varying structures (not just strings or numbers)

Accessing Children

Any data/content placed between the opening and closing tags of a component is accessible via this.props.children.

```
<CustomComponent>
  <p>This is child content</p>
</CustomComponent>
```

Inside CustomComponent:

```
render() {
  return <div>{this.props.children}</div>;
}
```

Example: BorderWrap Component

```
class BorderWrap extends React.Component {
  render() {
    const borderedStyle = { border: "1px solid silver", padding: 6 };
    return (
      <div style={borderedStyle}>
        {this.props.children}
      </div>
    );
  }
}
```

Usage:

```
<BorderWrap>
  <ExampleComponent />
</BorderWrap>
```

Updating IssueRow to Use Children

You can embed JSX or plain text within an IssueRow, like this:

```
<IssueRow issue_id={1}>Error in console when clicking Add</IssueRow>
<IssueRow issue_id={2}>
  <div>Missing <b>bottom</b> border on panel</div>
</IssueRow>
```

Update the IssueRow to use both props and children:

```
class IssueRow extends React.Component {
  render() {
    const style = this.props.rowStyle;
    return (
      <tr>
        <td style={style}>{this.props.issue_id}</td>
        <td style={style}>{this.props.issue_title}</td>
```

```
      <td style={style}>{this.props.children}</td>
    </tr>
  );
 }
}
```

Dynamic Composition (React Components)

What is Dynamic Composition?

Dynamic composition refers to the practice of **programmatically generating React components** (like <IssueRow />) from a JavaScript array, instead of hardcoding them manually. This approach is scalable and maintainable, especially when dealing with large or dynamic data sets.

Step-by-Step Breakdown

1. Create an In-Memory Array of Issues

We store a few issue objects in a globally declared array called issues. Each object includes various fields such as id, status, owner, effort, created, due, and title.

```
const issues = [
  {
    id: 1, status: 'New', owner: 'Ravan', effort: 5,
    created: new Date('2018-08-15'), due: undefined,
    title: 'Error in console when clicking Add',
  },
  {
    id: 2, status: 'Assigned', owner: 'Eddie', effort: 14,
    created: new Date('2018-08-16'), due: new Date('2018-08-30'),
    title: 'Missing bottom border on panel',
  },
];
```

Note: We left the due field undefined in one object to test optional field rendering.

2. Modify the IssueTable to Generate Rows Dynamically

Instead of hardcoding each row, use Array.map() to transform each issue object into an <IssueRow /> component.

```
const issueRows = issues.map(issue =>
  <IssueRow key={issue.id} issue={issue} />
);
```

And render it inside the table like this:

```
<tbody>
  {issueRows}
</tbody>
```

You can also directly write the map expression inside JSX:

```
<tbody>
  {issues.map(issue => <IssueRow key={issue.id} issue={issue} />)}
</tbody>
```

This shows that JSX allows any valid JavaScript *expression* inside {}.

3. Add Table Header and Styling

Define a <thead> section and assign a className to the table for CSS styling.

```
<table className="bordered-table">
```

Include the following CSS in index.html to style the table:

```
<style>
  table.bordered-table th, td {
    border: 1px solid silver;
    padding: 4px;
  }
  table.bordered-table {
```

```
    border-collapse: collapse;
  }
</style>
```

4. Update IssueRow to Use Full Object

Instead of passing each field as a prop, the entire issue object is passed:

```
<IssueRow issue={issue} />
```

Inside IssueRow, we extract the data and display it in <td> elements:

```
class IssueRow extends React.Component {
  render() {
    const issue = this.props.issue;
    return (
      <tr>
        <td>{issue.id}</td>
        <td>{issue.status}</td>
        <td>{issue.owner}</td>
        <td>{issue.created.toDateString()}</td>
        <td>{issue.effort}</td>
        <td>{issue.due ? issue.due.toDateString() : ''}</td>
        <td>{issue.title}</td>
      </tr>
    );
```

```
  }
}
```

Important Concepts

Concept	Explanation
key Prop	Required in a list of components to uniquely identify each item for React's rendering optimization. Here, issue.id is used.
Dynamic Rendering	JSX supports JavaScript expressions, so map() can be used inside JSX to render multiple components.
Object Prop Passing	Passing the entire issue object is cleaner and reduces boilerplate compared to passing each property individually.
Optional Fields	Use the ternary operator (? :) to safely render optional fields like due.

Summary

- We dynamically composed components (<IssueRow />) using the map() function.
- We styled our table using CSS classes.
- We used props to pass an entire object to a component.
- We handled optional fields safely using conditional rendering.
- We learned how JSX enables JavaScript expressions directly in markup.