VISVESVARAYA TECHNOLOGICAL UNIVERSITY BELGAUM

FULL STACK DEVELOPMENT

(Subject Code: BIS601)

LECTURE NOTES

VI-SEMESTER

Dr. Lokesh M R

Professor, Dept of ISE

A J INSTITUTE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

(A unit of Laxmi Memorial Education Trust. (R))

NH - 66, KottaraChowki, Kodical Cross - 575 006

VISION AND MISSION STATEMENT OF AJIET

Vision of the Institute

To produce top-quality engineers who are groomed for attaining excellence in their profession and competitive enough to help in the growth of nation and global society.

Mission of the Institute

- To offer affordable high-quality graduate program in engineering with value education and make the students socially responsible.
- To support and enhance the institutional environment to attain research excellence in both faculty and students and to inspire them to push the boundaries of knowledge base.
- To identify the common areas of interest amongst the individuals for the effective industry-institute partnership in a sustainable way by systematically working together.
- To promote the entrepreneurial attitude and inculcate innovative ideas among the engineering professionals.

Department of Information Science and Engineering

Vision of the Department

To be a center of excellence in Information Science & Engineering education, research and training to meet the growing needs of the industry and society

Mission of the Department

M 1. To impart theoretical and practical knowledge through the concepts and technologies in Information Science and Engineering

M 2. To foster research, collaboration and higher education with premier institutions and industries.

M 3. Promote innovation and entrepreneurship to fulfill the needs of the society and industry.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)	
PEO1	Analyse, design and implement solutions to the real-world problems in the field of Information Science and Engineering with multidisciplinary setup
PEO2	Keep abreast with the technology, innovation and pursue higher education with high standards of social and professional ethics
PEO3	Develop professional and entrepreneurship skills to work effectively as an individual and in a team to meet the ever-changing goals of the organization

PROGRAM OUTCOMES (POs)	
PO1	**Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and specialization in Mechanical Engineering for the solution of complex engineering problems.
PO2	**Problem analysis**: Identify, formulate, research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools, including prediction and modelling to complex engineering activities, with an understanding of the limitations.
PO6	**The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice
PO7	**Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication**: Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project Management and Finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environment.
PO12	**Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)	
PSO1	Design, implement and maintain the information systems that fulfill the current needs of the industry and society.
PSO2	Apply computational theory, storage and networking concepts to solve the day to day problems of the world

COURSE SYLLABUS

Course outcomes

At the end of the course the student will be able to:

CO1	Apply Javascript to build dynamic and interactive Web projects .
CO2	Implement user interface components for JavaScript-based Web using React.JS
CO3	Apply Express/Node to build web applications on the server side.
CO4	Develop data model in an open source nosql database.
CO5	Demonstrate modularization and packing of the front-end modules .

Suggested Learning Resources:

1. Text Books

2. 1. Jon Duckett, "JavaScript & jQuery: Interactive Front-End Web Development", Wiley, 2014.
3. 2. Vasan Subramanian, Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node. Apress, 2019.

2. Reference Books

1. NIL

4. Web links and Video Lectures (e-Resources):

1. https://github.com/vasansr/pro-mern-stack
2. https://nptel.ac.in/courses/106106156
3. https://archive.nptel.ac.in/courses/106/105/106105084/

1.8 VTU Syllabus

FULL STACK DEVELOPMENT (Effective from the academic year 2024 -2025) SEMESTER– VI			
Course Code	BIS601	**CIE Marks**	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	**SEE Marks**	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
Credits	04	**Exam Hours**	03
Examination nature (SEE)	Theory/Practical		

Course Learning Objectives: This course (BCS702) will enable students to:

1. To understand the essential javascript concepts for web development
2. To style Web applications using bootstrap.
3. To utilize React JS to build front end User Interface.
4. To understand the usage of API's to create web applications using Express JS.
5. To store and model data in a no sql database.

Teaching-Learning Process (General Instructions)

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.

1. Lecturer method (L) need not to be only a traditional lecture method, but alternative effective teaching methods could be adopted to attain the outcomes.
2. Use of Video/Animation to explain functioning of various concepts.
3. Encourage collaborative (Group Learning) Learning in the class.
4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes critical thinking.
5. Adopt Problem Based Learning (PBL), which fosters students' Analytical skills, develop design thinking skills such as the ability to design, evaluate, generalize, and analyze information rather than simply recall it.
6. Introduce Topics in manifold representations.
7. Show the different ways to solve the same problem with different circuits/logic and encourage the students to come up with their own creative ways to solve them.
8. Discuss how every concept can be applied to the real world - and when that's possible, it helps improve the students' understanding.

Module 1	**Contact Hours**
Basic JavaScript Instructions, Statements, Comments, Variables, Data Types, Arrays, Strings, Functions, Methods & Objects, Decisions & Loops. **Text Book 1: Chapter 2, 3,**	8
Module 2	
Document Object Model: DOM Manipulation, Selecting Elements, Working with DOM Nodes, Updating Element Content & Attributes, Events, Different Types of Events, How to Bind an Event to an Element, Event Delegation, Event Listeners. **Text Book 1: Chapter: 5, 6, 13**	8

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. **Text Book 2: Chapter 1, 2, 3**	8
Module 4	
React State: Initial State, Async State Initialization, Updating State, Lifting State Up, Event Handling, Stateless Components, Designing Components, State vs. Props, Component Hierarchy, Communication, Stateless Components. **Express**, REST API, GraphQL, Field Specification, Graph Based, Single Endpoint, Strongly Typed, Introspection, Libraries, The About API GraphQL Schema File, The List API, List API Integration, Custom Scalar types, The Create API, Create API Integration, Query Variables, Input Validations, Displaying Errors. **Text Book 2: Chapter 4, 5**	8
Module 5	
MongoDB: Basics, Documents, Collections, Databases, Query Language, Installation, The Mongo Shell, MongoDB CRUD Operations, Create, Read, Projection, Update, Delete, Aggregate, MongoDB Node.js Driver, Schema Initialization, Reading from MongoDB, Writing to MongoDB. **Text Book 2: Chapter 6, 7**	8

Course Outcomes(Course Skill Set)**:** At the end of the course, the student will be able to:

1. Apply Javascript to build dynamic and interactive Web projects .
2. Implement user interface components for JavaScript-based Web using React.JS
3. Apply Express/Node to build web applications on the server side.
4. Develop data model in an open source nosql database.
5. Demonstrate modularization and packing of the front-end modules .

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50) ●

IPCC means practical portion integrated with the theory of the course.

● CIE marks for the theory component are 25 marks and that for the practical component is 25 marks. ● 25 marks for the theory component are split into 15 marks for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and 10 marks for other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.

● Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for 25 marks).

● The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

CIE for the practical component of the IPCC

● 15 marks for the conduction of the experiment and preparation of laboratory record, and 10 marks for the test to be conducted after the completion of all the laboratory sessions.

● On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.

● The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to 15 marks.

● The laboratory test (duration 02/03 hours) after completion of all the experiments shall be conducted for 50 marks and scaled down to 10 marks.

● Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for 25 marks.

● The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.

SEE for IPCC :**Question Paper Pattern:**

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (duration 03 hours)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), should have a mix of topics under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored by the student shall be proportionally scaled down to 50 Marks

Textbooks: Suggested Learning Resources:

1. Jon Duckett, "JavaScript & jQuery: Interactive Front-End Web Development", Wiley, 2014.
2. Vasan Subramanian, Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node. Apress, 2019.

Reference Books:

Web links and Video Lectures (e-Resources):

- https://github.com/vasansr/pro-mern-stack
- https://nptel.ac.in/courses/106106156
- https://archive.nptel.ac.in/courses/106/105/106105084/

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

● Course Project: Build Web applications using MERN stack. Students (group of 2) can choose any real world problem from domains such as finance, marketing, medical, or enterprise projects (25 marks).

FULL STACK DEVELOPMENT LABORATORY
(Effective from the academic year 2024 -2025)
SEMESTER– VI

Course Code	BIS601	**CIE Marks**	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	**SEE Marks**	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100

PRACTICAL COMPONENT OF IPCC

Course Learning Objectives: This course (18CSL38) will enable students to:

This laboratory course enable students to get practical experience in design, develop, implement, analyze and evaluation/testing of

1. Apply Javascript to build dynamic and interactive Web projects .
2. Implement user interface components for JavaScript-based Web using React.JS
3. Apply Express/Node to build web applications on the server side.
4. Develop data model in an open source nosql database.
5. Demonstrate modularization and packing of the front-end modules .

Descriptions (if any):

- Implement all the programs in „C / C++“ Programming Language and Linux / Windows as OS.

Programs List:

1.	a. Write a script that Logs "Hello, World!" to the console. Create a script that calculates the sum of two numbers and displays the result in an alert box. b. Create an array of 5 cities and perform the following operations: Log the total number of cities. Add a new city at the end. Remove the first city. Find and log the index of a specific city.
2.	a. Read a string from the user, Find its length. Extract the word "JavaScript" using substring() or slice(). Replace one word with another word and log the new string. Write a function isPalindrome(str) that checks if a given string is a palindrome (reads the same backward).
3.	Create an object student with properties: name (string), grade (number), subjects (array), displayInfo() (method to log the student's details) Write a script to dynamically add a passed property to the student object, with a value of true or false based on their grade. Create a loop to log all keys and values of the student object.
4.	Create a button in your HTML with the text "Click Me". Add an event listener to log "Button clicked!" to the console when the button is clicked. Select an image and add a mouseover event listener to change its border color. Add an event listener to the document that logs the key pressed by the user.
5.	Build a React application to track issues. Display a list of issues (use static data). Each issue should have a title, description, and status (e.g., Open/Closed). Render the list using a functional component.
6	Create a component Counter with A state variable count initialized to 0. Create Buttons to increment and decrement the count. Simulate fetching initial data for the Counter component using useEffect (functional component) or componentDidMount (class component). Extend the Counter component to Double the count value when a button is clicked. Reset the count to 0 using another button.
7	Install Express (npm install express). Set up a basic server that responds with "Hello, Express!" at the root endpoint (GET /). Create a REST API. Implement endpoints for a Product resource: GET : Returns a list of products. POST : Adds a new product. GET /:id: Returns details of a specific product. PUT /:id: Updates an existing product. DELETE /:id: Deletes a product. Add middleware to log requests to the console. Use express.json() to parse incoming JSON payloads.
8	Install the MongoDB driver for Node.js. Create a Node.js script to connect to the shop database. Implement insert, find, update, and delete operations using the Node.js MongoDB driver. Define a product schema using Mongoose. Insert data into the products collection using Mongoose. Create an Express API with a /products endpoint to fetch all products. Use fetch in React to call the /products endpoint and display the list of products. Add a POST /products endpoint in Express to insert a new product. Update the Product List, After adding a product, update the list of products displayed in React.

1 Module

Introduction

Basic JavaScript Instructions, Statements, Comments, Variables, Data Types, Arrays, Strings, Functions, Methods & Objects, Decisions & Loops.

Text Book 1: Chapter 2, 3,

Module -1 Lecturer Notes:

Basic JavaScript Instructions

This Lecture Notes provides a comprehensive overview of Introduction, features, and history of JavaScript, Parallel hardware and parallel software, based on Chapter 1, and 2 of Jon Duckett, "JavaScript & jQuery: Interactive Front-End Web Development", Wiley, 2014 – Basic JavaScript Instructions, Statements, Comments, Variables, Data Types, Arrays, Strings, Functions, Methods & Objects, Decisions & Loops

MODULE-1: Introduction, features, and history of JavaScript

1. Basic JavaScript Instructions,
2. Statements,
3. Comments,
4. Variables,
5. Data Types,
6. Arrays,
7. Strings,
8. Functions,
9. Methods & Objects,
10. Decisions & Loops.

Text Book 1: Chapter 2, 3,

1. Introduction, features, and history of JavaScript
2. Writing JavaScript (inline, internal, external)
3. Statements and control flow (if, switch, loops)
4. Variables (`const`, `let`, `var`)
5. Data types (primitive & non-primitive)
6. Arrays and array methods
7. String methods
8. Objects, methods, `this`, ES6 features
9. Decision making and basic error handling

1. Introduction to JavaScript

JavaScript is a **lightweight, interpreted, and high-level scripting language** primarily used to make **web pages interactive and dynamic**. It executes on the **client side** within a web browser and enables functionalities such as form validation, dynamic content manipulation, event handling, and animations.

JavaScript works in close integration with **HTML** (structure) and **CSS** (presentation), forming the **core technologies of web development**. With the introduction of **Node.js**, JavaScript is also used for **server-side programming**, making it a key language in **full stack development**.

Features of JavaScript

The important features of JavaScript as per VTU syllabus are:

1. **Client-Side Scripting**
   JavaScript runs in the browser, reducing server workload and improving application responsiveness.
2. **Interpreted Language**
   JavaScript code is executed line by line without prior compilation, allowing faster development and debugging.
3. **Event-Driven Programming**
   JavaScript responds to user actions such as mouse clicks, key presses, and page loads.
4. **Object-Oriented Support**
   Supports objects, properties, methods, and prototypes for modular and reusable code.
5. **Platform Independent**
   JavaScript programs can run on different browsers and operating systems.
6. **Loosely Typed Language**
   Variables are not bound to a specific data type, enabling flexible programming.
7. **Easy Integration with HTML and CSS**
   JavaScript can be embedded directly into HTML pages and can manipulate HTML elements dynamically.

History of JavaScript

- **1995**: JavaScript was developed by **Brendan Eich** at Netscape and was initially named **Mocha**.
- **1996**: It was renamed **JavaScript** and released with **Netscape Navigator**.
- **1997**: JavaScript was standardized as **ECMAScript (ES1)** by **ECMA International**.
- **2009**: **ECMAScript 5 (ES5)** introduced improved syntax, strict mode, and better browser compatibility.
- **2015**: **ECMAScript 6 (ES6 / ES2015)** introduced modern features such as let, const, arrow functions, classes, and modules.
- **Present**: Continuous updates (ES7, ES8, ESNext) enhance performance, security, and functionality.

Key Points

- JavaScript is a **client-side, interpreted scripting language**.
- It supports **event-driven and object-oriented programming**.
- Standardized as **ECMAScript** to ensure cross-browser compatibility.
- ES6 marked a **major milestone** in modern JavaScript development.

2. JavaScript (Inline, Internal, External)

JavaScript code can be written and included in a web page using **three different methods**: **Inline JavaScript, Internal JavaScript, and External JavaScript**. These methods provide flexibility in how JavaScript is applied to HTML documents.

1. Inline JavaScript

Inline JavaScript is written **directly inside HTML elements** using event attributes such as onclick, onmouseover, etc.

Syntax Example

```
<button onclick="alert('Hello, World!')">Click Me</button>
```

Explanation

- JavaScript code is embedded within the HTML tag.
- It executes when a specific event occurs (e.g., button click).

Advantages

- Simple and easy for small scripts
- Useful for quick testing or demonstrations

Disadvantages

- Not suitable for large applications
- Makes HTML code lengthy and difficult to maintain
- Violates separation of concerns

2. Internal JavaScript

Internal JavaScript is written inside the <script> tag within an HTML file, usually placed in the <head> or at the end of the <body> section.

Syntax Example

```
<script>
   document.write("Hello, JavaScript!");
</script>
```

Explanation

- JavaScript code is placed within the HTML document itself.
- Suitable when the script is used only in one webpage.

Advantages

- Better organization than inline JavaScript
- Easy to debug for small projects

Disadvantages

- Cannot be reused across multiple pages
- Increases the size of HTML file

3. External JavaScript

External JavaScript is written in a **separate .js file** and linked to the HTML document using the <script> tag with the src attribute.

JavaScript File (script.js**)**

```
console.log("Hello, JavaScript!");
```

HTML File

```
<script src="script.js"></script>
```

Explanation

- JavaScript code is stored in a separate file.
- The browser loads and executes the external script.

Advantages

- Promotes code reusability
- Improves maintainability and readability
- Faster page loading due to browser caching
- Best practice for large and professional applications

Disadvantages

- Requires additional HTTP request (minimal impact)

Key Points

- JavaScript can be written in **three ways**: Inline, Internal, and External.
- **Inline JS** is written inside HTML tags.
- **Internal JS** is written inside the <script> tag.
- **External JS** is written in a separate .js file and linked using src.
- **External JavaScript is recommended** for large-scale applications.

3. JavaScript Statements

JavaScript **statements** are instructions that the browser executes to perform specific tasks. A JavaScript program consists of a **sequence of statements** executed **from top to bottom**, unless the flow is altered by control statements such as conditions or loops.

Types of JavaScript Statements

1. Declaration Statements
2. Assignment Statements
3. Expression Statements
4. Control Flow Statements
5. Looping Statements
6. Function Statements
7. Error Handling Statements

Control Flow Statements

Control flow statements determine the **order in which statements are executed** based on conditions.

1. if Statement

The if statement executes a block of code **only if a given condition is true**.

Syntax

```
if (condition) {
   // code to execute
}
```

Example

```
let age = 18;
if (age >= 18) {
   console.log("You are eligible to vote.");
}
```

2. if–else Statement

The if–else statement executes one block of code if the condition is true and another block if it is false.

Syntax

```
if (condition) {
   // true block
} else {
   // false block
}
```

Example

```
let num = 10;
if (num % 2 === 0) {
```

```
  console.log("Even number");
} else {
  console.log("Odd number");
}
```

3. if–else–if Statement

Used when **multiple conditions** need to be checked sequentially.

Example

```
let marks = 85;

if (marks >= 90) {
  console.log("Grade: A+");
} else if (marks >= 75) {
  console.log("Grade: A");
} else {
  console.log("Grade: B");
}
```

4. switch Statement

The switch statement evaluates an expression and executes code blocks based on **matching case values**. It is an alternative to long if–else chains.

Syntax

```
switch (expression) {
  case value1:
    // code
    break;
  case value2:
    // code
    break;
  default:
    // code
}
```

Example

```
let day = 3;
let dayName;

switch (day) {
  case 1: dayName = "Monday"; break;
  case 2: dayName = "Tuesday"; break;
  case 3: dayName = "Wednesday"; break;
  default: dayName = "Invalid day";
}

console.log(dayName);
```

Key Points

- Uses **strict equality (===)**
- break prevents fall-through
- default executes if no case matches

JavaScript Loops

Loops are used to **repeat a block of code** as long as a specified condition is true, reducing code redundancy.

1. for Loop

Used when the **number of iterations is known**.

Syntax

```
for (initialization; condition; increment/decrement) {
  // code
}
```

Example

```
for (let i = 1; i <= 3; i++) {
  console.log("Count:", i);
}
```

2. while Loop

Executes code **as long as the condition is true**.

Syntax

```
while (condition) { // code
}
```

Example

```
let i = 0;
while (i < 3) {
  console.log("Number:", i);
  i++;
}
```

3. do–while Loop

Executes the loop body **at least once**, then checks the condition.

Syntax

```
do {
  // code
} while (condition);
```

Example

```
let i = 0;
do {
  console.log("Iteration:", i);
  i++;
} while (i < 3);
```

4. for–in Loop

Used to iterate over the **properties (keys) of an object**.

Example

```
const obj = { name: "Ashish", age: 25 };

for (let key in obj) {
  console.log(key + ":", obj[key]);
}
```

5. for–of Loop

Used to iterate over **iterable objects** such as arrays and strings.

Example

```
let arr = [1, 2, 3, 4];
for (let value of arr) {
  console.log(value);
}
```

Break and Continue Statements

break

Terminates the loop immediately when a condition is met.

```
for (let i = 0; i < 10; i++) {
  if (i === 5) break;
  console.log(i);
}
```

continue

Skips the current iteration and continues with the next one.

```
for (let i = 1; i <= 10; i++) {
  if (i % 2 === 0) continue;
  console.log(i);
}
```

Key Point

- Control flow statements manage **execution order**
- if, if–else, switch handle **decision making**
- Loops (for, while, do–while) handle **repetition**
- break exits a loop, continue skips an iteration
- for–in → objects, for–of → arrays/iterables

4. Variables in JavaScript

A **variable** is a named memory location used to store data values that can be **referenced and manipulated** throughout a program.

JavaScript is a **loosely typed (dynamically typed) language**, meaning variables do not require explicit data type declaration.

A variable must be **declared before it is used**.

Ways to Declare Variables in JavaScript

JavaScript provides **three keywords** for declaring variables:

1. var
2. let
3. const

1. const (Constant Variables)

The const keyword is used to declare variables whose **reference cannot be reassigned**.

Syntax

```
const a = 10;
```

Key Characteristics

- Must be **initialized at the time of declaration**
- Cannot be **reassigned**
- Block-scoped
- Preferred in modern JavaScript

Example

```
const PI = 3.14;
// PI = 3.1415;  // Error: Assignment to constant variable
```

Important Note

const does **not** mean the value is immutable.

If a const variable refers to an **object or array**, its contents **can be modified**, but the reference cannot change.

```
const arr = [1, 2, 3];
arr.push(4);   // Allowed
```

2. let (Block Scoped Variables)

The `let` keyword is used for variables whose values **can be reassigned**.

Syntax

```
let x = 5;
x = 10;
```

Key Characteristics

- Block-scoped
- Can be **reassigned**
- Can be **declared first and initialized later**
- Cannot be redeclared in the same scope

Example

```
let count;
count = 10;
```

3. var (Function Scoped Variables)

Before ES6 (2015), `var` was the only way to declare variables in JavaScript.

Syntax

```
var name = "JavaScript";
```

Key Characteristics

- Function-scoped
- Can be redeclared
- Can cause unexpected bugs due to hoisting
- **Not recommended** in modern JavaScript

Example

```
var a = 10;
var a = 20; // Allowed (but not recommended)
```

Multiple Variable Declaration

```
const a = 1, b = 2;
let x = 10, y = 20;
```

Best Practices

- Prefer using **const by default**
- Use **let only when reassignment is required**
- Avoid using **var** in modern JavaScript
- Reduces bugs and improves code reliability

Key Points

- JavaScript variables are **dynamically typed**
- const → cannot be reassigned
- let → block-scoped and reassignable
- var → function-scoped and outdated
- ES6 introduced let and const

5. JavaScript Data Types (Primitive and Non-Primitive)

Data Types in JavaScript

A **data type** specifies the kind of value a variable can hold.

JavaScript is a **dynamically typed (loosely typed) language**, meaning the data type of a variable is determined **at runtime** and can change during execution.

JavaScript data types are broadly classified into:

1. **Primitive Data Types**
2. **Non-Primitive (Reference) Data Types**

1. Primitive Data Types

Primitive data types store **single, simple values** and are **immutable** (cannot be altered).

Types of Primitive Data Types

1. Number

Represents both integer and floating-point numbers.

```
let a = 10;
let b = 3.14;
```

2. String

Represents a sequence of characters enclosed in **single (`' '`), double (`" "`) or backticks ( )**.

```
let name = "JavaScript";
```

3. Boolean

Represents logical values: `true` or `false`.

```
let isActive = true;
```

4. Undefined

A variable declared but **not assigned** a value.

```
let x;
```

5. Null

Represents an **intentional absence of value**.

```
let value = null;
```

6. Symbol (ES6)

Used to create **unique identifiers**.

```
let sym = Symbol("id");
```

7. BigInt

Used to represent very large integers beyond the Number limit.

```
let big = 123456789012345678901n;
```

Summary of Primitive Data Types

Data Type	Description
Number	Numeric values
String	Textual data
Boolean	true / false
Undefined	Declared but not assigned
Null	Empty or no value
Symbol	Unique identifiers
BigInt	Large integers

2. Non-Primitive (Reference) Data Types

Non-primitive data types can store **multiple values** and are **mutable**. They are stored by **reference**, not by value.

1. Object

An object stores data in **key–value pairs**.

```
let student = {
  name: "Anil",
  age: 20
};
```

2. Array

An array is an ordered collection of elements.

```
let marks = [85, 90, 78];
```

3. Function

A block of reusable code.

```
function add(a, b) {
  return a + b;
}
```

4. Date

Used to work with dates and time.

```
let today = new Date();
```

Examples of Non-Primitive Data Types

Type	Description
Object	Collection of properties
Array	Ordered list of values
Function	Reusable block of code
Date	Date and time representation

Difference Between Primitive and Non-Primitive Data Types

Feature	Primitive	Non-Primitive
Value storage	By value	By reference
Mutability	Immutable	Mutable
Complexity	Simple	Complex
Examples	Number, String	Object, Array

Type Checking in JavaScript

The `typeof` operator is used to find the data type.

```
typeof 10;         // number
typeof "Hello";    // string
typeof true;       // boolean
```

VTU Exam Key Points

- JavaScript is a **dynamically typed language**
- Data types are classified into **primitive and non-primitive**
- Primitive types store **single values**
- Non-primitive types store **collections of values**
- Objects and arrays are **reference types**

6. Arrays in JavaScript

An **array** is a non-primitive (reference) data type used to store **multiple values** in a **single variable**. Array elements are stored in **indexed positions**, starting from index **0**.

JavaScript arrays can store **different data types** in the same array.

Declaring Arrays

1. Using Array Literal (Preferred)

```
let numbers = [10, 20, 30, 40];
```

2. Using Array Constructor

```
let colors = new Array("Red", "Green", "Blue");
```

Accessing and Modifying Array Elements

```
let marks = [85, 90, 78];
marks[1] = 95;       // Modify element
console.log(marks[0]); // Access element
```

Array Properties

length

Returns the number of elements in an array.

```
let arr = [1, 2, 3];
console.log(arr.length); // 3
```

Common Array Methods (VTU Important)

1. push()

Adds one or more elements to the **end** of the array.

```
let arr = [1, 2];
arr.push(3);
```

2. pop()

Removes the **last element** from the array.

```
arr.pop();
```

3. unshift()

Adds elements to the **beginning** of the array.

```
arr.unshift(0);
```

4. shift()

Removes the **first element** from the array.

```
arr.shift();
```

5. concat()

Merges two or more arrays and returns a new array.

```
let a = [1, 2];
let b = [3, 4];
let c = a.concat(b);
```

6. join()

Joins array elements into a **string**.

```
let names = ["A", "B", "C"];
names.join("-");
```

7. slice()

Extracts a portion of an array **without modifying** the original array.

```
let nums = [10, 20, 30, 40];
nums.slice(1, 3);
```

8. splice()

Adds or removes elements **by modifying** the original array.

```
nums.splice(1, 1, 25);
```

9. indexOf()

Returns the index of the specified element.

```
nums.indexOf(30);
```

10. includes()

Checks whether an element exists in an array.

```
nums.includes(20);
```

11. sort()

Sorts array elements.

```
nums.sort();
```

12. reverse()

Reverses the array order.

```
nums.reverse();
```

Iterating Through Arrays

Using for Loop

```
for (let i = 0; i < nums.length; i++) {
    console.log(nums[i]);
}
```

Using forEach()

```
nums.forEach(function(value) {
    console.log(value);
});
```

Difference Between slice() and splice()

Feature	slice()	splice()
Modifies original array	✗ No	✓ Yes
Returns	New array	Removed elements
Use	Extract elements	Add/remove elements

Key Points

- Arrays store **multiple values in a single variable**
- JavaScript arrays are **dynamic and heterogeneous**
- Index starts from **0**
- push() and pop() work at the **end**
- shift() and unshift() work at the **beginning**
- slice() does not modify the array
- splice() modifies the array

7.Strings in JavaScript

A **string** is a primitive data type used to represent **textual data**.
Strings can be created using:

- Single quotes ''
- Double quotes ""
- Backticks ``` (template literals)

```
let str = "JavaScript";
```

Strings in JavaScript are **immutable**, meaning their contents **cannot be changed directly**. Any operation on a string returns a **new string**.

Common String Methods

1. length

Returns the number of characters in a string.

```
let s = "Hello";
console.log(s.length); // 5
```

2. toUpperCase()

Converts the string to **uppercase**.

```
"hello".toUpperCase(); // "HELLO"
```

3. toLowerCase()

Converts the string to **lowercase**.

```
"HELLO".toLowerCase(); // "hello"
```

4. charAt()

Returns the character at a specified index.

```
"JavaScript".charAt(4); // 'S'
```

5. indexOf()

Returns the **first occurrence index** of a specified value.

```
"JavaScript".indexOf("S"); // 4
```

6. lastIndexOf()

Returns the **last occurrence index** of a specified value.

```
"JavaScript".lastIndexOf("a"); // 3
```

7. substring()

Extracts characters between two indices.

```
"JavaScript".substring(0, 4); // "Java"
```

8. slice()

Extracts part of a string and supports **negative indices**.

```
"JavaScript".slice(-6); // "Script"
```

9. replace()

Replaces a specified value with another value.

```
"Hello World".replace("World", "JavaScript");
```

10. split()

Splits a string into an array based on a delimiter.

```
"HTML,CSS,JS".split(",");
```

11. trim()

Removes whitespace from **both ends** of a string.

```
"  Hello  ".trim();
```

12. startsWith()

Checks whether a string starts with a specified value.

```
"JavaScript".startsWith("Java");
```

13. endsWith()

Checks whether a string ends with a specified value.

```
"JavaScript".endsWith("Script");
```

14. includes()

Checks whether a string contains a specified value.

```
"JavaScript".includes("Script");
```

String Concatenation

Using + Operator

```
let a = "Hello";
let b = "World";
let c = a + " " + b;
```

Using Template Literals

```
let name = "VTU";
let msg = `Welcome to ${name}`;
```

Difference Between slice(), substring() and substr()

Feature	slice()	substring()
Negative index	✓ Yes	✗ No
Extracts part	✓ Yes	✓ Yes
Usage	Modern JS	Traditional

Key Points

- Strings are **immutable**
- length gives total characters
- toUpperCase() / toLowerCase() change case
- slice() supports negative index
- split() converts string to array
- trim() removes extra spaces
- Template literals use **backticks**

8.Objects in JavaScript

An **object** is a non-primitive data type used to store **multiple related values** in the form of **key–value pairs (properties)**.

Objects represent real-world entities and help organize code efficiently.

Creating Objects

1. Using Object Literal (Most Common)

```
let student = {
    name: "Anil",
    usn: "1AJ21IS001",
    marks: 85
};
```

2. Using new Object()

```
let emp = new Object();
emp.id = 101;
emp.name = "Ravi";
```

Accessing Object Properties

```
student.name;         // Dot notation
student["marks"];    // Bracket notation
```

Object Methods

A **method** is a function defined inside an object.

Example

```
let person = {
    name: "Rahul",
    greet: function() {
        return "Hello " + this.name;
    }
};
```

The this Keyword

The this keyword refers to the **current object** that is executing the method.

Example

```
let car = {
    brand: "Toyota",
    getBrand: function() {
        return this.brand;
    }
};
```

Key Points

- this refers to the object **calling the method**
- Behavior depends on **execution context**
-

ES6 (ECMAScript 2015) Features

ES6 introduced several features to make JavaScript **more powerful, readable, and efficient**.

Important ES6 Features

1. let and const

- `let` → block-scoped, reassignable
- `const` → block-scoped, not reassignable

```
let x = 10;
const y = 20;
```

2. Arrow Functions

Provide a shorter syntax for functions.

```
const add = (a, b) => a + b;
```

3. Template Literals

Allow embedding variables in strings using backticks.

```
let name = "VTU";
let msg = `Welcome to ${name}`;
```

4. Default Parameters

Allows function parameters to have default values.

```
function greet(name = "User") {
    return "Hello " + name;
}
```

5. Destructuring Assignment

Extracts values from arrays or objects.

```
let arr = [1, 2];
let [a, b] = arr;
let obj = {x: 10, y: 20};
let {x, y} = obj;
```

6. Spread Operator (...)

Expands elements of arrays or objects.

```
let nums = [1, 2];
let newNums = [...nums, 3, 4];
```

7. Rest Parameters

Allows variable number of arguments.

```
function sum(...args) {
    return args.reduce((a, b) => a + b);
}
```

8. Classes

Provide blueprint for creating objects.

```
class Student {
    constructor(name) {
        this.name = name;
    }
}
```

9. Modules

Used to export and import code.

```
export default Student;
import Student from "./Student.js";
```

Difference Between Function and Arrow Function

Feature	Function	Arrow Function
Syntax	Traditional	Short
`this` binding	Dynamic	Lexical
Use case	General	Callbacks

Key Points

- Objects store data in **key–value pairs**
- Methods are **functions inside objects**
- this refers to the **current object**
- ES6 introduced let, const, arrow functions, classes
- ES6 improves **code readability and maintainability**

9.Decision Making in JavaScript

Decision making allows a program to execute **different blocks of code** based on specified conditions. JavaScript uses **conditional statements** to control the flow of execution.

Decision Making Statements

1. if Statement

Executes a block of code when the condition is **true**.

```
let age = 20;
if (age >= 18) {
    console.log("Eligible to vote");
}
```

2. if–else Statement

Executes one block if the condition is true and another if it is false.

```
let num = 7;
if (num % 2 === 0) {
    console.log("Even number");
} else {
    console.log("Odd number");
}
```

3. if–else–if Ladder

Used to check **multiple conditions** sequentially.

```
let marks = 80;

if (marks >= 90) {
    console.log("Grade A+");
} else if (marks >= 75) {
    console.log("Grade A");
} else {
    console.log("Grade B");
}
```

4. switch Statement

Used when multiple conditions depend on the **same expression**.

```
let choice = 2;

switch (choice) {
    case 1:
        console.log("Option One");
        break;
    case 2:
        console.log("Option Two");
        break;
    default:
        console.log("Invalid Option");
}
```

Decision Making -Key Points

- `if` → single condition
- `if-else` → two outcomes
- `if-else-if` → multiple conditions
- `switch` → alternative to long if–else ladder
- `break` prevents fall-through

Basic Error Handling in JavaScript

Error handling helps manage **runtime errors** and prevents abnormal termination of programs.
JavaScript provides **exception handling** using:

- `try`

- `catch`
- `finally`
- `throw`

1. try–catch Block

Used to test a block of code for errors.

```
try {
    let result = x / 10;
    console.log(result);
} catch (error) {
    console.log("Error occurred");
}
```

2. finally Block

Executes **regardless of error occurrence**.

```
try {
    let a = 10 / 0;
} catch (err) {
    console.log("Error");
} finally {
    console.log("Execution completed");
}
```

3. throw Statement

Used to create **custom errors**.

```
let age = -5;

if (age < 0) {
    throw "Invalid age";
}
```

Common JavaScript Errors

Error Type	Description
Syntax Error	Mistake in code syntax
Reference Error	Variable not defined
Type Error	Invalid operation on data type

Key Points

- Decision making controls **program flow**
- if, if–else, switch used for conditions
- Error handling prevents program crashes
- try–catch handles runtime errors
- finally executes in all cases
- throw generates custom errors

2 Module

Document Object Model: DOM Manipulation, Selecting Elements, Working with DOM Nodes, Updating Element Content & Attributes, Events, Different Types of Events, How to Bind an Event to an Element, Event Delegation, Event Listeners.

Text Book 1: Chapter: 5, 6, 13

Module -2 Lecturer Notes: Document Object Model:

This Lecture Notes provides Chapter: 5, 6, 13 of Jon Duckett, "JavaScript & jQuery: Interactive Front-End Web Development", Wiley, 2014 to comprehensive overview of Introduction to DOM, Document Object Model: DOM Manipulation, Selecting Elements, Working with DOM Nodes, Updating Element Content & Attributes, Events, Different Types of Events, How to Bind an Event to an Element, Event Delegation, Event Listeners.

MODULE-2

1. Document Object Model: DOM Manipulation,
2. Selecting Elements,
3. Working with DOM Nodes,
4. Updating Element Content & Attributes,
5. Events,
6. Different Types of Events,
7. How to Bind an Event to an Element,
8. Event Delegation,
9. Event Listeners.

Text Book 1: Chapter: 5, 6, 13

MODULE-2: Introduction to DOM

1. Introduction to DOM

- DOM represents an HTML document as a tree of objects.
- JavaScript and jQuery can dynamically access and manipulate elements.

2. Selecting Elements

- **JavaScript:** getElementById, getElementsByClassName, getElementsByTagName, querySelector, querySelectorAll
- **jQuery:** $() using CSS-like selectors

3. Modifying Elements

- Content: innerHTML, textContent, .html(), .text()
- Attributes: setAttribute, .attr()

4. Adding, Removing, Replacing Nodes

- Create: document.createElement()
- Add: appendChild(), .append(), .prepend()
- Remove/Replace: removeChild(), .remove(), .replaceWith()
- Clone: cloneNode(true/false), .clone()

5. DOM Nodes & Navigation

- Node types: element, attribute, text, comment
- Navigation: parentNode, children, firstChild, nextSibling
- jQuery equivalents: .parent(), .children(), .next()

6. Event Handling

- **JavaScript:** addEventListener(), removeEventListener()
- **jQuery:** .on(), .off()
- Event types: mouse, keyboard, form, window, clipboard, drag-drop, media, touch

7. Event Propagation

- Bubbling and capturing phases
- event.stopPropagation()

8. Event Delegation

- Handling dynamic elements efficiently using parent listeners
- Implemented using event.target (JS) or delegated .on() (jQuery)

9. Styling, Effects & AJAX

- CSS manipulation: element.style, .css()
- jQuery effects: fadeIn, fadeOut, slideUp, animate
- Data fetching: Fetch API and $.ajax()
-

1. Introduction to DOM

The **DOM (Document Object Model)** is a programming interface that represents an **HTML document as a hierarchical tree of objects**. In this tree structure, every part of the web page—such as elements, attributes, and text—is treated as a **node**.

Using the DOM, **JavaScript can access, modify, add, or delete HTML elements dynamically** without reloading the page. This makes web pages interactive and responsive to user actions like clicks, keyboard input, or form submissions.

In simple terms, the DOM acts as a **bridge between HTML and JavaScript**, allowing scripts to control the structure, content, and style of a web page at runtime.

2. Selecting Elements in DOM

Selecting elements in the DOM is the process of identifying and accessing specific HTML elements so that they can be manipulated using JavaScript or jQuery. The PDF explains two main approaches:

1. Using JavaScript

JavaScript provides several built-in methods to select elements based on different criteria:

- `document.getElementById("id")` – Selects a single element using its unique **ID**.
- `document.getElementsByClassName("class")` – Selects all elements that share a **class name**.
- `document.getElementsByTagName("tag")` – Selects elements based on their **HTML tag**.
- `document.querySelector("selector")` – Selects the **first element** that matches a CSS selector.
- `document.querySelectorAll("selector")` – Selects **all elements** matching a CSS selector.
-

2. Using jQuery

- `$("selector")` – Selects elements using **CSS-style selectors**, making element selection simpler and more concise.

Overall, selecting elements is a **fundamental DOM operation** that enables developers to read, modify, style, and attach events to web page components dynamically.

3. Modifying Elements

Modifying elements in the DOM refers to changing the **content or attributes** of existing HTML elements dynamically using JavaScript or jQuery. This allows web pages to update their display and behavior without reloading.

Using JavaScript

- `element.innerHTML = "New Content";`
  → Changes the HTML content inside an element (including tags).
- `element.textContent = "New Text";`
  → Updates only the text inside an element (ignores HTML tags).
- `element.setAttribute("attribute", "value");`
  → Modifies or adds an attribute of an element.

Using jQuery

- `$("selector").html("New Content");`
  → Changes the inner HTML of the selected element.
- `$("selector").text("New Text");`
  → Updates only the text content.
- `$("selector").attr("attribute", "value");`
  → Sets or updates an attribute.

In summary, modifying elements enables **dynamic content updates, attribute changes, and interactive behavior** in web applications, making it a core concept of DOM manipulation.

4. Adding, Removing, and Replacing Nodes

Adding, removing, and replacing nodes in the DOM allows developers to dynamically change the structure of a web page by creating new elements, deleting existing ones, or substituting one element with another.

Using JavaScript

- `document.createElement("tag")`
  → Creates a new HTML element (node).
- `parent.appendChild(child)`
  → Adds a child node to a parent element.
- `parent.insertBefore(newNode, existingNode)`
  → Inserts a new node before an existing node.
- `parent.removeChild(child)`
  → Removes a specified child node from the parent.
- `parent.replaceChild(newNode, oldNode)`
  → Replaces an existing node with a new node.

Example:

A new element can be created using document.createElement(), assigned content, and then added to the document using appendChild().

Using jQuery

- `$("parent").append("<tag>New Content</tag>")`
  → Adds content at the end of the selected element.
- `$("parent").prepend("<tag>New Content</tag>")`
  → Adds content at the beginning.
- `$("selector").remove()`
  → Removes the selected element.
- `$("selector").replaceWith("<new element>")`
  → Replaces an element with new content.

In essence, these operations make web pages **dynamic and interactive**, enabling real-time updates to page structure based on user actions or application logic.

5. DOM Nodes & Navigation

A **DOM node** is any individual object in the Document Object Model tree. The PDF explains that every part of an HTML document is treated as a node, which enables structured access and manipulation.

Types of DOM Nodes

- **Element nodes** – HTML elements such as `<div>`, `<p>`, `<button>`
- **Attribute nodes** – Attributes like `id`, `class`, `style`
- **Text nodes** – The textual content inside elements
- **Comment nodes** – HTML comments (`<!-- comment -->`)

Accessing and Navigating Nodes

Using JavaScript

- `element.parentNode` – Accesses the parent of an element
- `element.children` – Returns all child elements
- `element.firstChild` / `element.lastChild` – Gets the first or last child
- `element.nextSibling` / `element.previousSibling` – Navigates between sibling nodes

Example:

```
let parent = document.getElementById("child").parentNode;
let firstChild = document.getElementById("parent").firstChild;
```

Using jQuery

- `$("selector").parent()` – Selects the parent element
- `$("selector").children()` – Selects child elements
- `$("selector").first()` – Selects the first child
- `$("selector").next()` – Selects the next sibling

Example:

```
let parent = $("#child").parent();
let firstChild = $("#parent").children().first();
```

In summary, **DOM nodes and navigation methods** allow developers to move through the document tree efficiently, making it possible to manipulate parent, child, and sibling elements dynamically in web applications.

6. Event Handling

Event handling refers to the process of responding to user actions such as mouse clicks, key presses, form submissions, window resizing, and other interactions on a web page. The PDF explains event handling using both **JavaScript** and **jQuery**.

Using JavaScript

- `element.addEventListener("event", function);`
  This method attaches an event listener to an element and executes a function when the specified event occurs.
- It supports **multiple event listeners** on the same element and does **not overwrite existing handlers**.

Example:

```
document.getElementById("btn").addEventListener("click", function() {
    alert("Button Clicked!");
});
```

Using jQuery

- `$("selector").on("event", function() { });`
  The `.on()` method is used to bind events in a simpler and more flexible way.
- It works efficiently with **dynamically added elements** and can handle multiple events.

Example:

```
$("#btn").on("click", function() {
    alert("Button Clicked!");
});
```

Common Events Mentioned

- Mouse events (click, mouseover)
- Keyboard events (keydown)
- Form events (submit, change)
- Window events (resize)
- Touch, drag-and-drop, media, and clipboard events

In conclusion, event handling enables **interactive and responsive web applications** by allowing JavaScript and jQuery to detect and respond to user actions in real time.

7. Event Propagation

Event propagation describes the way an event travels through the DOM hierarchy when it is triggered on an element. The PDF explains that events move through elements in **two main phases**:

1. Capturing Phase (Top to Bottom)

- The event starts from the **root of the DOM** and moves down toward the target element.
- This phase is less commonly used in practical applications.

2. Bubbling Phase (Bottom to Top)

- The event starts from the **target element** and then propagates upward to its parent elements.
- This is the **default behavior** in most browsers.

Example of Event Bubbling:

```
document.getElementById("parent").addEventListener("click", function() {
    alert("Parent Clicked!");
});

document.getElementById("child").addEventListener("click", function(event) {
    alert("Child Clicked!");
});
```

When the child element is clicked, both the **child** and **parent** event handlers are triggered due to bubbling.

Stopping Event Propagation

- `event.stopPropagation()` is used to prevent the event from moving further up the DOM tree.

Example:

```
document.getElementById("child").addEventListener("click", function(event) {
    event.stopPropagation();
    alert("Child Clicked, but parent won't trigger!");
});
```

In summary, **event propagation** explains how events flow through the DOM, and understanding it is essential for handling complex interactions, especially when working with **nested elements and event delegation**.

8. Event Delegation

Event delegation is a technique in JavaScript and jQuery where a **single event listener is attached to a parent element** instead of attaching event listeners to multiple child elements. This parent listener handles events triggered by its child elements using event propagation (bubbling).

Why Event Delegation is Used

- **Efficient:** Reduces memory usage by minimizing the number of event listeners.
- **Handles dynamic elements:** Works even for elements added to the DOM after page load.
- **Better performance:** Especially useful for large lists, tables, or repetitive UI components.

How Event Delegation Works

1. An event listener is added to a common **parent element**.
2. When an event occurs, it bubbles up to the parent.
3. event.target is used to identify the actual child element that triggered the event.
4. The handler executes only if the target matches the required condition.

Using JavaScript

```
document.getElementById("parent").addEventListener("click", function(event) {
    if (event.target.matches(".child")) {
        console.log("Child clicked!");
    }
});
```

Using jQuery

```
$("#parent").on("click", ".child", function() {
    console.log("Child clicked!");
});
```

When to Use Event Delegation

- When working with **lists, tables, or containers** having many child elements
- When elements are **created dynamically**
- When optimizing performance for **high-interaction web pages**

When Not to Use Event Delegation

- If child elements require **different or unique event handlers**
- If events like focus, blur, or mouseenter are needed
- When immediate response is critical in very large DOM structures

In conclusion, **event delegation** is a powerful and efficient event-handling technique that leverages event bubbling to manage both static and dynamic elements effectively.

9. Styling, Effects & AJAX

The section **"Styling, Effects & AJAX"** explains how web pages can be made visually dynamic and data-driven using **JavaScript and jQuery**.

1. Changing CSS Styles

Using JavaScript

- `element.style.property = "value";` → Dynamically changes the CSS style of an element.

Example:

```
document.getElementById("box").style.backgroundColor = "blue";
```

Using jQuery

- `$("selector").css("property", "value");` → Modifies CSS styles using a simple syntax.

Example:

```
$("#box").css("background-color", "blue");
```

2. Animations & Effects in jQuery

jQuery provides built-in effects to create smooth animations and transitions:

- fadeIn() – Gradually displays an element
- fadeOut() – Gradually hides an element
- slideUp() – Slides an element upward
- slideDown() – Slides an element downward
- animate({property: "value"}, duration) – Creates custom animations

These effects improve **user experience and visual interaction**.

3. AJAX and Fetching Data

Using JavaScript Fetch API

- Used to fetch data from a server asynchronously without reloading the page.

```
fetch("url")
  .then(response => response.json())
  .then(data => console.log(data));
```

Using jQuery AJAX

- `$.ajax()` is used for asynchronous server communication.

```
$.ajax({
  url: "url",
  method: "GET",
  success: function(data) {
    console.log(data);
  }
});
```

Summary

This section highlights how **styling** enhances appearance, **effects and animations** improve interactivity, and **AJAX** enables dynamic data loading, together forming the foundation of modern interactive web applications.

3 Module

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.

Text Book 2: Chapter 1, 2, 3

Module -3 Lecturer Notes: Introduction to MERN

This Lecture Notes provides a comprehensive overview of Introduction to MERN, based on Chapter 4 and 5 of " Vasan Subramanian, Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node. Apress, 2019.– An Introduction to Parallel Programming, second edition, Morgan Kauffman , VTU syllabus based BCS601 Full stack development" It is designed to serve as a detailed lecture presentation for a classroom setting, covering 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.

Text Book 2: Chapter 1, 2, 3

MODULE-3: Introduction to MERN

1. Form enhancement and validation.
2. Introduction to MERN: MERN components,
3. Server less Hello world.
4. React Components: Issue Tracker,
5. React Classes,
6. Composing Components,
7. Passing Data Using Properties,
8. Passing Data Using Children,
9. Dynamic Composition.

Text Book 2: Chapter 1, 2, 3

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 the 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 the 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.

Everything in React is a component:

- Components are independent, reusable, and maintain their own state.

3. Component-Based Architecture

- Components 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**.
- Doesn't rely on threads — uses **callbacks** and an **event loop**.

Node.js – Event Driven Architecture

- Uses asynchronous, non-blocking I/O.
- 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 the **MERN stack**.
- A **NoSQL, document-oriented database**.

What is MongoDB?

- 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 and columns.
- **MongoDB** stores data in **objects/documents**, removing this mismatch.
- Data is stored as **documents (JSON-like)**.
- In relational databases, this mismatch requires **ORMs** (e.g., **Hibernate**).
- Leads to a need for **complex mappings**.

Document-Oriented Storage

- 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.
- Enables 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 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.

ReactDOM: Converts React components to actual DOM elements.

Tools Used:

React: JavaScript library for building UI.

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>:

const element = React.createElement('div', { title: 'Outer div' });

This is where your React component will render.

Create a React element using React.createElement():

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
  <script                                                    src="https://unpkg.com/react-
dom@16/umd/reactdom.development.js"></script>
</head>
<body>
  <div id="content"></div>
  <script>     const element = React.createElement('div', {title: 'Outer div'},
     React.createElement('h1', null, 'Hello World!')

src="https://unpkg.com/react@16/umd/react.development.js"></script>
    );
    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

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

This adds Express as a dependency in package.json.

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?

- Reusability
- JSX alone is limited — real-world apps need interactivity and data handling

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)
- **created**: auto-set date
- **status**: open / assigned / resolved, etc. (list)
- **due**: optional due date (date)
- **effort**: estimated days (number)

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

extends React.Component {}.

- Example:

You can define a React class component using class ComponentName

```
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**.

✓ **Filter issues**

✓ **See a table of issues**

✓ **Add a new issue**

For example, imagine a page where users can:

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 IssueAdd extends React.Component {
  render() {
    return <div>This is a placeholder for adding an issue.</div>;
  }
}

class IssueTable extends React.Component {
  render() {
    return <div>This is a placeholder for the issue table.</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.

Project Structure

```
myexpressapp/
├── backend/          # Your Express.js backend (if any)
├── frontend/         # Your React frontend
│   ├── src/
│   │   ├── 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.

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

- Example Use Case: 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}
         />
         <IssueRow
           rowStyle={rowStyle}
           issue_id={2}
           issue_title="Error in console when clicking Add"
         />
         <IssueRow
           rowStyle={rowStyle}
           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.

Using `children` in React Components

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`.

Example:

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

Inside `CustomComponent`:

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

Example: `BorderWrap` Component

```
const borderedStyle = {
  border: '1px solid silver',
  padding: 6
};
```

```
class BorderWrap extends React.Component {
  render() {
    return (
      <div style={borderedStyle}>
        {this.props.children}
      </div>
    );
  }
}
```

Usage:

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

Missing `<b>bottom</b>` border on panel

```
<div>Missing <b>bottom</b> border on panel</div>
```

Update `IssueRow` to Use Both Props and Children

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}>
</IssueRow>
```

`IssueRow` Component Code

```
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 in React

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`
- `title`

```
const issues = [
  {
    id: 1,
    status: 'New',
    owner: 'Ravan',
    effort: 5,
    created: new Date(2018, 0, 15),
    due: undefined,
    title: 'Error in console when clicking Add',
  },
  {
    id: 2,
    status: 'Assigned',
    owner: 'Eddie',
    effort: 14,
    created: new Date(2018, 0, 1),
    due: new Date(2018, 0, 3),
    title: 'Missing bottom border on panel',
  },
];
```

Note:
We left the `due` field **undefined** in one object to test an optional field.

2. 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 Props Passing	Passing the entire `issue` object is cleaner and reduces boilerplate compared to passing each property individually.
Optional Fields like `due`	Use the ternary operator (`? :`) to safely render optional fields.

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.

4 Module

React State –React State: Initial State, Async State Initialization, Updating State, Lifting State Up, Event Handling, Stateless Components, Designing Components, State vs. Props, Component Hierarchy, Communication, Stateless Components.Express, REST API, GraphQL, Field Specification, Graph Based, Single Endpoint, Strongly Typed, Introspection, Libraries, The About API GraphQL Schema File, The List API, List API Integration, Custom Scalar types, The Create API, Create API Integration, Query Variables, Input Validations, Displaying Errors.

Module -4 Lecturer Notes: React State

This Lecture Notes provides a comprehensive overview of **React State**, based on Chapter 4 and 5 of " Vasan Subramanian, Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node. Apress, 2019–, VTU syllabus based BIS601 Full stack Development" It is designed to serve as a detailed lecture presentation for a classroom setting, covering **React State**: Initial State, Async State Initialization, Updating State, Lifting State Up, Event Handling, Stateless Components, Designing Components, State vs. Props, Component Hierarchy, Communication, Stateless Components.**Express**, REST API, GraphQL, Field Specification, Graph Based, Single Endpoint, Strongly Typed, Introspection, Libraries, The About API GraphQL Schema File, The List API, List API Integration, Custom Scalar types, The Create API, Create API Integration, Query Variables, Input Validations, Displaying Errors.

MODULE-4: React State

Introduction to React State

1. **Static vs Dynamic Components** ○ Previously, we worked with static components that did not change.
   - To make components responsive, React uses **state**.
2. **What is State?**
   - State is a data structure in a React component that **changes over time**.
   - Unlike **props**, which are immutable, **state** is mutable and affects the UI.
3. **State and Rendering** ○ The UI updates automatically when the **state changes**.
   - React **re-renders** the component when the state is modified.
4. **Goal of the Chapter** ○ Add a button that appends a new row to the issue list on click.
   - Learn how to manipulate state, handle events, and pass data between components.
5. **Initial Implementation** ○ Start by adding a row using a **timer** instead of user interaction.
   - Later, replace the timer with a **button and a form** for user input.
6. **Defining State in a Component** ○ The **state** is stored in this.state as an object with key-value pairs.
   - Example: this.state = { issues: initialIssues };

State should include only dynamic data affecting the UI.

7. **Choosing What to Store in State** ○ Store data that **affects rendering** and **changes dynamically**. ○ Example: The list of issues should be stored in the state.
   - **Do not store** static values like table border styles.
8. **Modifying State in IssueTable Component**
   - Store the initial issues in a variable called initialIssues.
   - Update render() to use this.state.issues:

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

9. Initializing State in Constructor

The constructor sets the initial state:

```
class IssueTable extends React.Component {
  constructor() {
    super();
    this.state = { issues: initialIssues };
  }
}
```

10. Next Steps

- Implement state updates using `setState()`.
- Introduce user interaction with a button and form input.

Initial State

○ The state of a component is captured in a variable called this.state in the component's class, which should be an object consisting of one or more key-value pairs, where each key is a state variable name and the value is the current value of that variable. ○ React does not specify what needs to go into the state, but it is useful to store in the state anything that affects the rendered view and can change due to any event. These are typically events generated due to user interaction.

For now, let's just use an array of issues as the one and only state of the component and use that array to construct the table of issues.

Thus, in the render() method of IssueTable, let's change the loop that creates the set of IssueRows to use the state variable called issues rather than the global array like this

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

As for the initial state, let's use a hard-coded set of issues and set it to the initial state. We already have a global array of issues; let's rename this array to `initialIssues`, just to make it explicit that it is only an initial set.

```
...
const initialIssues = [
  ...
];
...
```

Setting the initial state needs to be done in the constructor of the component. This can be done by simply assigning the variable `this.state` to the set of state variables and their values. Let's use the variable `initialIssues` to initialize the value of the state variable `issues` like this:

```
...
    this.state = { issues: initialIssues };
...
```

Note that we used only one state variable called issues.

We can have other state variables, for instance if we were showing the issue list in multiple pages, and we wanted to also keep the page number currently being shown as another state variable, we could have done that by adding another key to the object like page: 0. The set of all changes to use the state to render the view of IssueTable is shown in Listing 4-1

Listing 4-1. App.jsx: Initializing and Using State

```
...
const issues = [
const initialIssues = [
  {
    id: 1, status: 'New', owner: 'Ravan', effort: 5,
    created: new Date('2018-08-15'), due: undefined,
  },
...
class IssueTable extends React.Component {
  constructor() {
    super();
    this.state = { issues: initialIssues };
  }

  render() {
    const issueRows = issues.map(issue =>


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

Async State Initialization

1. State Initialization in the Constructor

- React **state must be assigned** in the constructor.
- Since data is fetched asynchronously, initialize issues as an **empty array**:

```
constructor() {
  super();
  this.state = { issues: [] };
}
```

2. Modifying State with setState()

- State updates must be done using this.setState().
- Example of updating the issues state dynamically: this.setState({ issues: newIssues });
- setState() **merges** the new state with the existing state.

Why Is an Async Call Needed for Fetching Data?

- Data is **usually fetched from a server**, not statically available.
- Fetching requires an **asynchronous API call**.
- We simulate this with setTimeout().

3. Simulating an API Call with setTimeout()

- Mimics real-world API call behavior:

```
loadData() {
  setTimeout(() => {
    this.setState({ issues: initialIssues });
  }, 500);
}
```

- The **500ms delay** represents an API response time.

4. Why Not Call loadData() in the Constructor?

- The **constructor only initializes** the component, but the UI rendering happens later.
- Calling setState() inside the constructor **may cause errors** if rendering is not finished.

5. React Lifecycle Methods for Managing State Updates

Lifecycle Method	Purpose	Can call `setState()`?
componentDidMount()	Called after the component is mounted to the DOM. Best place to fetch data.	✅ Yes
componentDidUpdate()	Called after an update occurs, except for the first render.	✅ Yes
componentWillUnmount()	Used for cleanup (e.g., canceling timers, aborting API calls).	❌ No
shouldComponentUpdate()	Used for optimizing renders by preventing unnecessary updates.	❌ No (returns `true/false`)

6. Fetching Data in componentDidMount()

- Ensures the **component is fully rendered** before making an API call.
- Implementation:

componentDidMount() { this.loadData();
}

Key Takeaways

- **State must be initialized in the constructor but updated with setState().**
- **Data fetching is asynchronous** and should be handled in lifecycle methods.
- **Use componentDidMount()** to safely fetch and update state.
- **Lifecycle methods** help manage state and optimize rendering.

Listing 4-2. App.jsx, IssueTable: Loading State Asynchronously

```
...
class IssueTable extends React.Component {
  constructor() {
    super();
    this.state = { issues: initialIssues };
    this.state = { issues: [] };
  }

  componentDidMount() {
    this.loadData();
  }

  loadData() {
    setTimeout(() => {
      this.setState({ issues: initialIssues });
    }, 500);
  }
...
```

If you refresh the browser (assuming you're still running npm run watch and npm start on two different consoles), you will find that the list of issue displayed as it used to be in the previous steps.But you will also see that for a fraction of a second after the page is loaded , the table is empty , as shown in fig 4.4

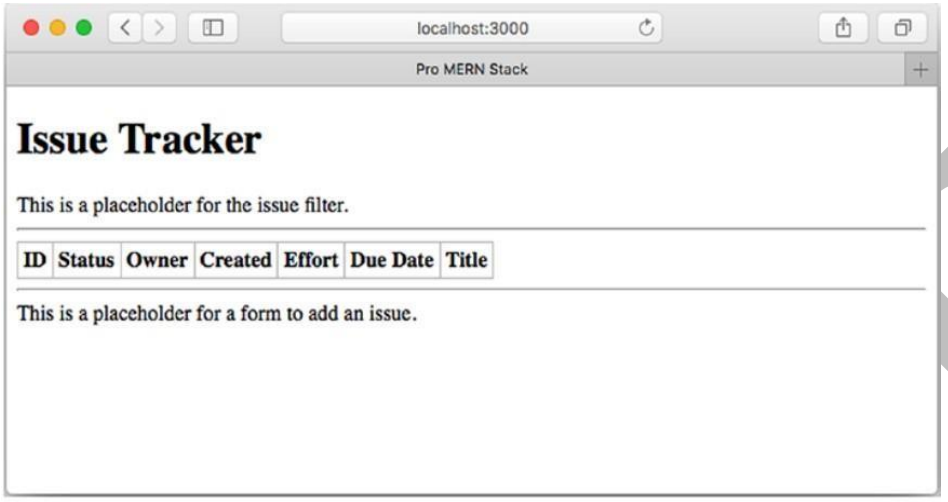

Figure 4-1. *Empty table shown for a fraction of a second*

Updating State

1. **Changing a Portion of the State Instead of Overwriting It**
   - Instead of replacing the whole state, we update just part of it.
   - Example: **Adding a new issue** to the existing issues array.
2. **Creating a Method to Add a New Issue**
   - The method assigns an **ID** and **creation date** before adding the issue:

```
createIssue(issue) {
  issue.id = this.state.issues.length + 1; issue.created = new Date();
}
```

3. **Directly Modifying State is NOT Allowed**
   - The state must be treated as **immutable**.
   - These **incorrect** examples modify state directly:

```
this.state.issues.push(issue);  // + Incorrect
issues = this.state.issues; issues.push(issue);
this.setState({ issues: issues }); // + Incorrect
```

4. **Why Can't We Modify State Directly?**
   - React **does not detect direct mutations** of the state.
   - Lifecycle methods that compare **previous and new states** may fail.
   - Using setState() ensures React detects changes and **rerenders** correctly.
5. **Correct Way: Using a Copy of the State**
   - Create a **shallow copy** of the array using slice():

```
const issues = this.state.issues.slice();
issues.push(issue); this.setState({ issues: issues });
```

   - This ensures React recognizes a **new reference** and updates the component properly.
6. **Alternative: Using Immutability Libraries**
   - **Libraries like Immutable.js** help manage deep state updates efficiently.
   - **Not needed for simple cases** like appending an issue.

7. Simulating an Automatic Issue Addition

- Instead of a UI, we **automatically add** an issue after 2 seconds using setTimeout().

8. Define a Sample Issue

- Declare a **predefined issue**:

```
const sampleIssue = {
  status: 'New', owner: 'Pieta',
};
```

9. Adding the Sample Issue with a Timer

```
title: 'Completion date should be optional',
setTimeout(() => {
  this.createIssue(sampleIssue);
}, 2000);
```

10. Key Takeaways

In constructor(), schedule an automatic addition of the issue:

Never modify state directly; always use setState().

Make a copy of the state before updating it.

Use lifecycle methods to manage state updates.

For complex state updates, consider using immutability libraries. State updates trigger React rerenders, ensuring UI consistency.

This should automatically add the sample issue to the list of issues after the page is loaded. The final set of changes—for using a timer to append a sample issue to the list of issues—is shown in Listing 4-3

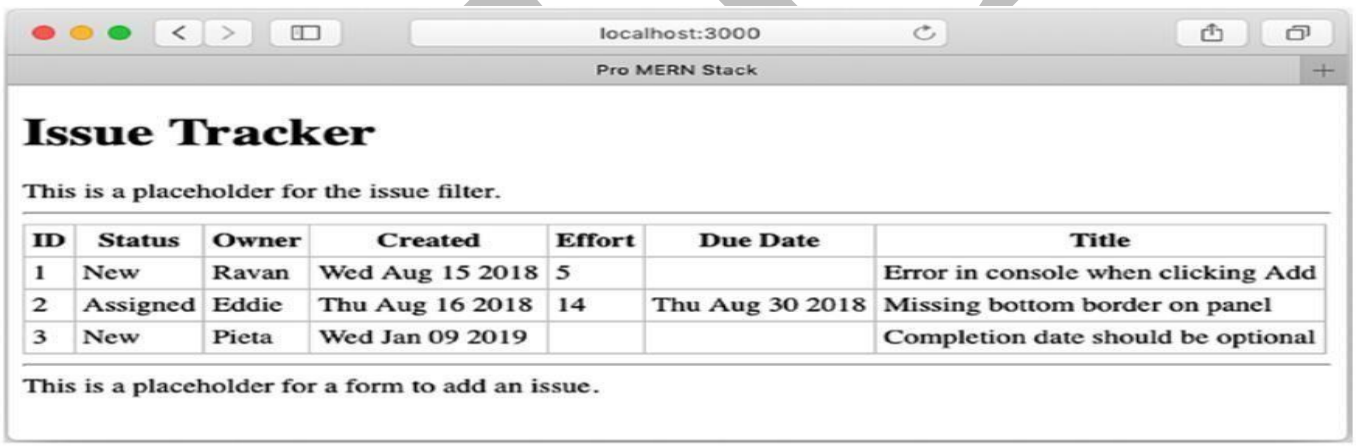

Figure 4-2. *Appended row to initial set of issues*

Listing 4-3. App.jsx: Appending an Issue on a Timer

```
...
const initialIssues = [
  ...
];

const sampleIssue = {
  status: 'New', owner: 'Pieta',
  title: 'Completion date should be optional',
};

...

class IssueTable extends React.Component {
  constructor() {
    super();
    this.state = { issues: [] };
    setTimeout(() => {
      this.createIssue(sampleIssue);
    }, 2000);
  }
```

```
...

createIssue(issue) {
  issue.id = this.state.issues.length + 1;
  issue.created = new Date();
  const newIssueList = this.state.issues.slice();
  newIssueList.push(issue);
  this.setState({ issues: newIssueList });
}
}
```

On running this set of changes and refreshing the browser, you'll see that there are two rows of issues to start with. After two seconds, a third row is added with a newly generated ID and the contents of the sample issue. A screenshot of the three- row table is shown in Figure 4-2

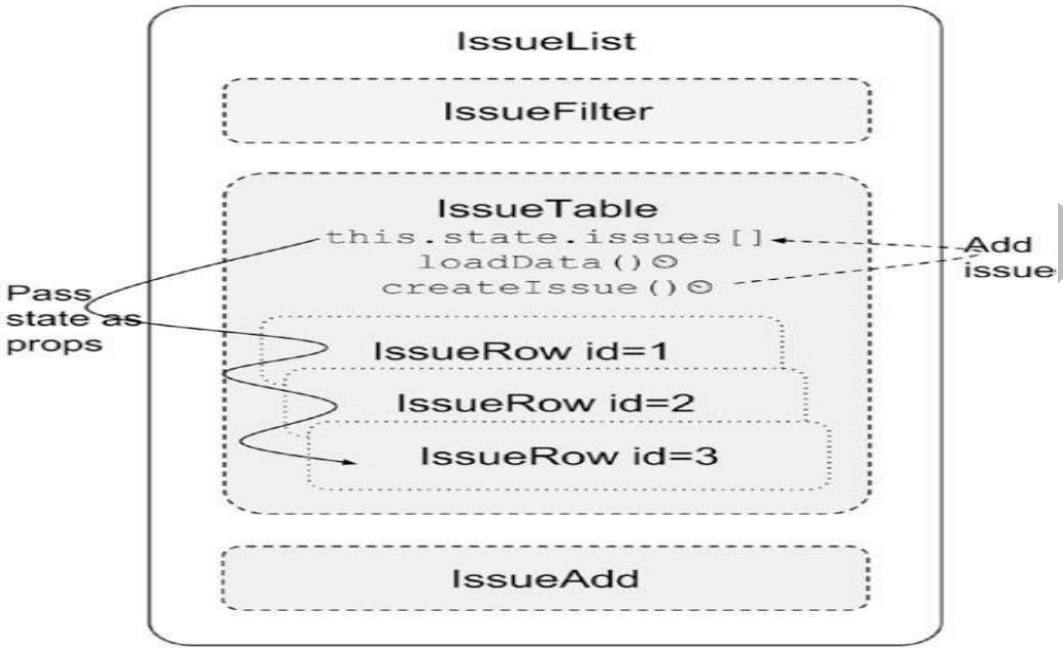

Figure 4-3. *Setting state and passing data as props*

Lifting State Up

In React, **Lifting State Up** is a pattern used to manage shared state between multiple components. Instead of keeping state separately in sibling components, we move it to their **closest common parent**. This allows child components to receive the shared state as **props** and update it via **callback functions** provided by the parent.

Why Lift State Up?

- **Ensures Sibling Communication:** React does not allow direct communication between sibling components. Instead, the parent can hold the state and pass it down.
- **Centralized State Management:** Keeping state in a common parent prevents data duplication and inconsistencies.
- **Easier Updates:** Since all state changes happen in the parent, updating and debugging become simpler.

Example (Issue Tracker Scenario)

- Initially, IssueTable managed the issues list and createIssue() method.
- To allow IssueAdd to trigger issue creation, **state and methods were moved to IssueList (the common parent).**
- IssueTable now receives issues as **props** instead of managing its own state. □ IssueAdd calls createIssue() through a **prop function** from IssueList.

Key Changes in Code

1. **Move State and Methods to Parent (IssueList)**

```
class IssueList extends React.Component {
  constructor() {
    super();
    this.state = { issues: [] };
    this.createIssue = this.createIssue.bind(this);
  }
  createIssue(issue) {
    issue.id = this.state.issues.length + 1; issue.created = new Date();
    const newIssueList = [...this.state.issues, issue];
    this.setState({ issues: newIssueList });
  } render() {
    return (
      <>
        <IssueTable issues={this.state.issues} />
        <IssueAdd createIssue={this.createIssue} /> </>
    );
  }
}
```

2. **Pass State Down as Props (IssueTable)**

```
<IssueTable issues={this.state.issues} />
```

3. Pass Method as Props (IssueAdd)

```
<IssueAdd createIssue={this.createIssue} />
```

4. Use the Passed Method in IssueAdd

```
this.props.createIssue(sampleIssue);
setTimeout(() => {
}, 2000);
```

This setup allows IssueAdd to trigger a new issue addition without directly modifying the state. Instead, it calls createIssue() in IssueList, ensuring state updates in a controlled way.

Event Handling

Let's now add an issue interactively, on the click of a button rather than use a timer to do this. We'll create a form with two text inputs and use the values that the user enters in them to add a new issue. An Add button will trigger the addition. Let's start by creating the form with two text inputs in the render() method of IssueAdd in place of the placeholder div

At this point, clicking Add will submit the form and fetch the same screen again. That's not what we want. Firstly, we want it to call createIssue() using the values in the owner and title fields. Secondly, we want to prevent the form from being submitted because we will handle the event ourselves.

So, let's rewrite the form declaration with a name and an on Submit handler like this.

Now, we can implement the method handleSubmit() in IssueAdd. This method receives the event that triggered the submit as an argument.In order to prevent the form from being submitted when the Add button is clicked, we need to call the preventDefault() func tion on the event.

After the call to createIssue(), let's keep the form ready for the next set of inputs by clearing the text input fields .

```
    ...
                <form name="issueAdd" onSubmit={this.handleSubmit}>
    ...
...
      <div>This is a placeholder for a form to add an issue.</div>
      <form>
        <input type="text" name="owner" placeholder="Owner" />
        <input type="text" name="title" placeholder="Title" />
        <button>Add</button>
      </form>
...
```

At this point, we can remove the timer that creates an issue from the constructor.

```
...
  constructor() {
    super();
    setTimeout(() => {
      this.props.createIssue(sampleIssue);
    }, 2000);
  }
...
```

If you run the code, you'll see a form being displayed in place of the placeholder in IssueAdd. The screenshot of how this looks is shown in Figure 4-4.

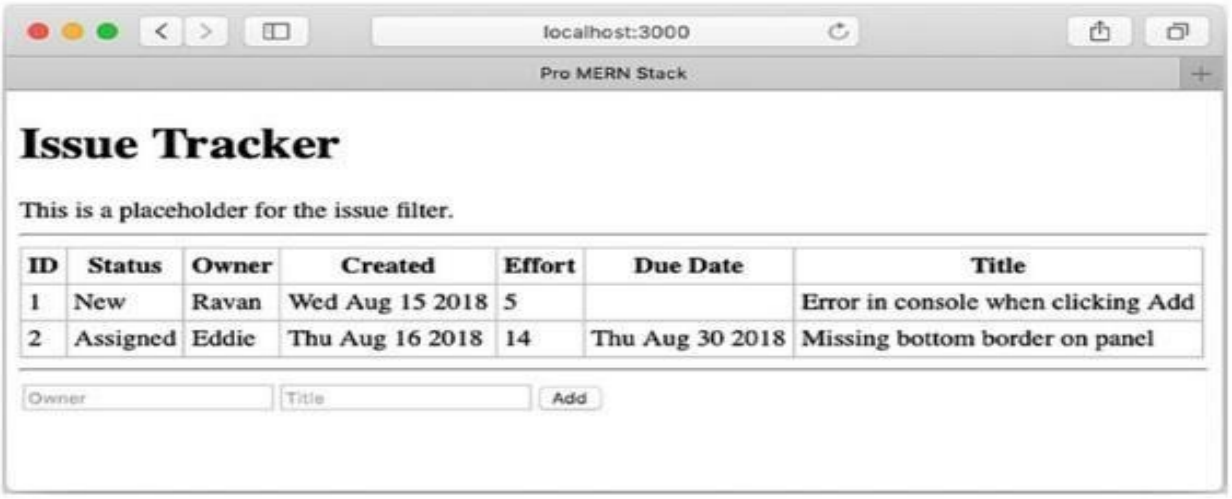

Figure 4-4. *IssueAdd placeholder replaced with a form*

```
...
  handleSubmit(e) {
    e.preventDefault();
    const form = document.forms.issueAdd;
    const issue = {
      owner: form.owner.value, title: form.title.value, status: 'New',
    }
    this.props.createIssue(issue);
    form.owner.value = ""; form.title.value = "";
  }
...
```

Since handleSubmit will be called from an event, the context, or this will be set to the object generating the event, which is typically the window object. Since handleSubmit will be called from an event, the context, or this will be set to the object generating the event, which is typically the window object.

```
...
  constructor() {
    super();
    this.handleSubmit = this.handleSubmit.bind(this);
  }
...
```

The new full code of the IssueAdd class, after these changes, is shown in Listing 4-7

Listing 4-7. App.jsx, IssueList: New IssueAdd Class

```
class IssueAdd extends React.Component {
  constructor() {
    super();
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleSubmit(e) {
    e.preventDefault();
    const form = document.forms.issueAdd;
    const issue = {
      owner: form.owner.value, title: form.title.value, status: 'New',
    }
    this.props.createIssue(issue);
    form.owner.value = ""; form.title.value = "";
  }

  render() {
    return (
      <form name="issueAdd" onSubmit={this.handleSubmit}>
        <input type="text" name="owner" placeholder="Owner" />
        <input type="text" name="title" placeholder="Title" />
        <button>Add</button>
      </form>
    );
  }
}
```

The global object sampleIssue is no longer required, so we can get rid of it. This change is shown in Listing 4-8.

Listing 4-8. App.jsx, Removal of sampleIssue

```
...
const sampleIssue = {
  status: 'New', owner: 'Pieta',
  title: 'Completion date should be optional',
};
...
```

You can now test the changes by entering some values in the owner and title fields and clicking Add. You can add as

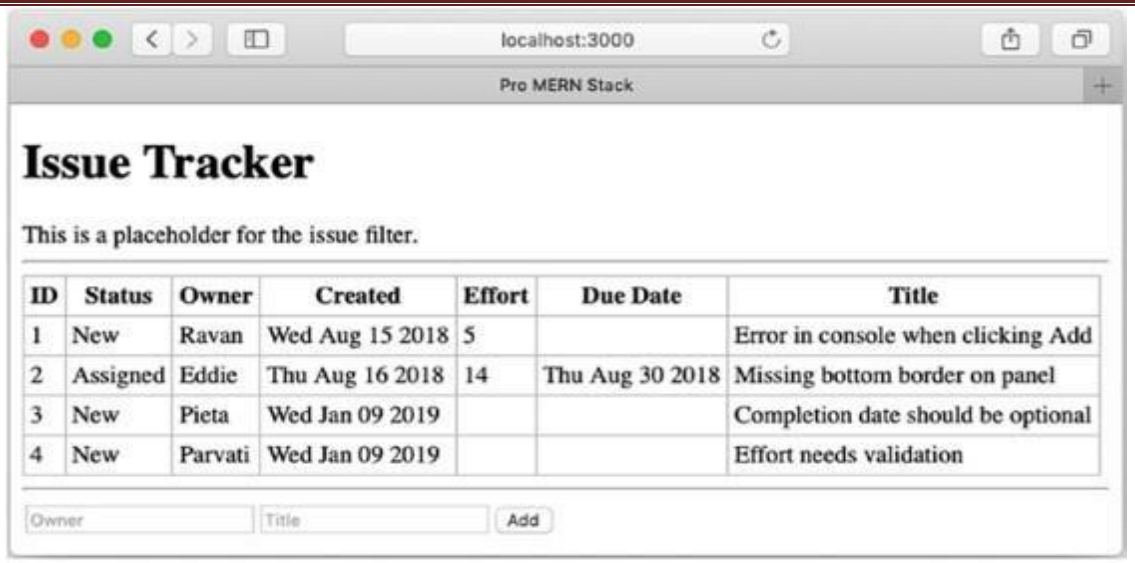

Figure 4-5. *Adding new issues using the IssueAdd form*

many rows as you like. If you add two issues, you'll get a screen like the one in Figure 4-5.

At the end of all this, we have been able to encapsulate and initiate the creation of a new issue from the IssueAdd component itself. This new UI hierarchy data and function flow is depicted in Figure 4-6.

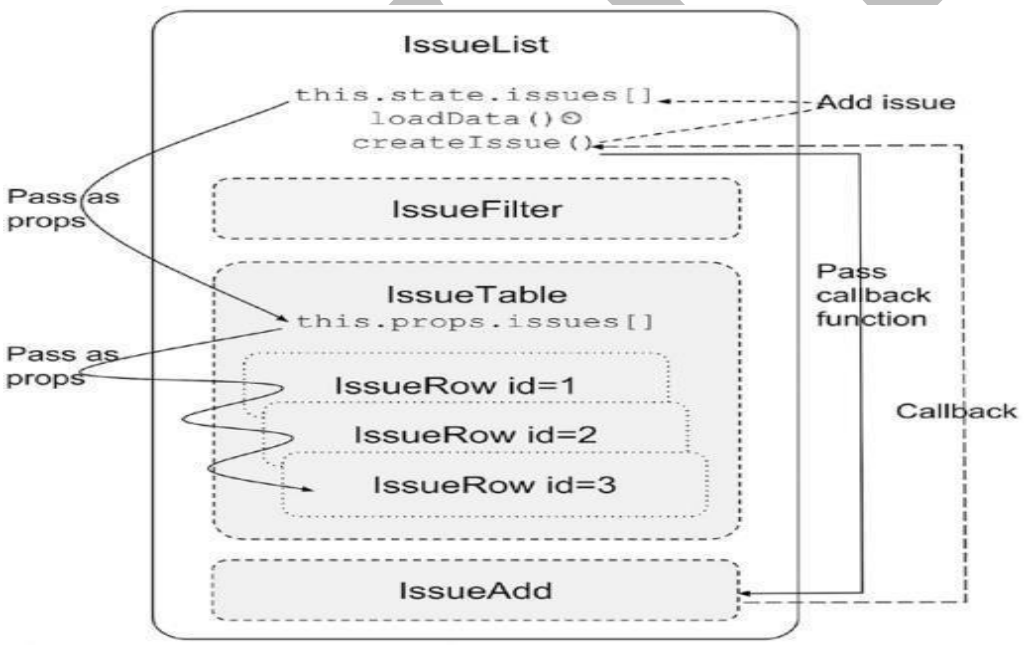

Figure 4-6. *Component hierarchy and data flow after lifting state up*

Stateless Components

Stateless components, also called **functional components**, are components that only receive props and render UI without maintaining any internal state.

Why Use Stateless Components?

- **Improved Performance:** They are faster because they don't have lifecycle methods or state updates.
- **Cleaner Code:** They are simpler and easier to read.
- **Better Maintainability:** They focus only on rendering, making debugging easier.

Example: Converting Class Components to Stateless Components Before (Class Component)

```
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>
    );
  }
}
```

After (Stateless Functional Component)

```
const IssueRow = ({ issue
}) => (
  <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>
);
```

Designing Components

Designing Components Most beginners will have a bit of confusion between state and props, when to use which, what granularity of components should one choose, and how to go about it all. This section is devoted to discussing some principles and best practices.

State vs. Prop

Both state and props hold model information, but they are different. The props are immutable, whereas state is not. Typically, state variables are passed down to child components as props because the children don't maintain or modify them. They take in a read-only copy and use it only to render the view of the component.

If any event in the child affects the parent's state, the child calls a method defined in the parent. Access to this method should have been explicitly given by passing it as a callback via props.

Anything that can change due to an event anywhere in the component hierarchy qualifies as being part of the state. Avoid keeping computed values in the state; instead, simply compute them when needed, typically inside the render() method.

You can use Table 4-1 as a quick reference to the differences.

Table 4-1. *State vs. Props*

Attribute	State	Props
Mutability	Can be changed using `this.setState()`	Cannot be changed
Ownership	Belongs to the component	Belongs to an ancestor, the component gets a read-only copy
Information	Model information	Model information
Affects	Rendering of the component	Rendering of the component

Component Hierarchy

- Split the application into components and subcomponents.
- Typically, this will reflect the data model itself. For example, in the Issue Tracker, the issues array was represented by the IssueTable component, and each issue was represented by the IssueRow component.
- Decide on the granularity just as you would for splitting functions and objects. The component should be self-contained with minimal and logical interfaces to the parent.
- If you find it doing too many things, just like in functions, it should probably be split into multiple components, so that it follows the Single Responsibility principle (that is, every component should be responsible for one and only one thing).
- If you are passing in too many props to a component, it is an indication that either the component needs to be split, or it need not exist: the parent itself could do the job

Communication

- Communication between components depends on the direction. Parents communicate to children via props; when state changes, the props automatically change.
- Children communicate to parents via callbacks. Siblings and cousins can't communicate with each other, so if there is a need, the information has to go up the hierarchy and then back down.
- This is called lifting the state up. This is what we did when we dealt with adding a new issue. The IssueAdd component had to insert a row in IssueTable.
- It was achieved by keeping the state in the least common ancestor, IssueList.
  The addition was initiated by IssueAdd and a new array element added in

IssueList's state via a callback.

- The result was seen in IssueTable by passing the issues array down as props from IssueList.
- If there is a need to know the state of a child in a parent, you're probably doing it wrong.
- Although React does offer a way using refs, you shouldn't feel the need if you follow the one-way data flow strictly: state flows as props into children, events cause state changes, which flows back as props.

Stateless Components

- In a well-designed application, most components would be stateless functions of their properties. All states would be captured in a few components at the top of the hierarchy, from where the props of all the descendants are derived.
- We did just that with the IssueList, where we kept the state. We converted all descendent components to stateless components, relying only on props passed down the hierarchy to render themselves.
- We kept the state in IssueList because that was the least common component above all the descendants that depended on that state.
- Sometimes, you may find that there is no logical common ancestor. In such cases, you may have to invent a new component just to hold the state, even though visually the component has nothing

Express

1. What is Express?

Express.js is a **minimal** and **flexible** web framework for Node.js. It provides essential web functionalities through **middleware** and enables efficient routing.

2. Routing in Express

Routing directs incoming HTTP requests to the appropriate handler.

Basic Route Example:

```
app.get('/hello', (req, res) => {
  res.send('Hello World!');
});
```

- The **HTTP method** (GET) and **path** (/hello) define the route.
- The **handler function** processes the request and sends a response.

Route Parameters:

Used to capture dynamic values from URLs.

```
app.get('/customers/:customerId', (req, res) => {
  res.send(`Customer ID: ${req.params.customerId}`);
});
```

- /customers/1234 → req.params.customerId = 1234

3. Express Request Matching and Order

- **Routes are matched in order** of definition.
- More specific routes should be defined **before** generic ones.

Example:

```
app.get('/api/issues', handler); // Specific app.use('/api/*', middleware); //
Generic (placed after)
```

4. Express Request Object (req)

Holds information about the HTTP request.

Property	Description
`req.params`	Captures route parameters (`/users/:id` → `req.params.id`)
`req.query`	Parses query strings (`/search?term=express` → `req.query.term`)
`req.header(name)`	Retrieves request headers (e.g., `Content-Type`)
`req.body`	Holds request body (used in `POST`, `PUT`, `PATCH`)

5. Express Response Object (res)

Used to send response back to the client

Method	Description
`res.send(body)`	Sends a response (text, buffer, or JSON).
`res.status(code)`	Sets HTTP status (`res.status(404).send('Not Found')`).
`res.json(object)`	Sends a JSON response.
`res.sendFile(path)`	Serves a file.

6. Middleware in Express

Middleware functions process requests before sending a response.

Application-Level Middleware:

```
app.use((req, res, next) => {
  console.log('Request received');
  next(); // Pass control to next middleware
});
```

Built-in Middleware:

- **express.static** → Serves static files (CSS, images, HTML).
- **Example:**
  app.use(express.static('public'));
  - Access files via /index.html, /style.css.

Path-Specific Middleware:

```
app.use('/public', express.static('public'));
```

- Static files must be accessed via /public/index.html.

REST API

1. What is REST?

REST (**Representational State Transfer**) is an **architectural pattern** for designing web APIs. It focuses on:

Resources (nouns, such as customers or orders)

HTTP methods (verbs like GET, POST, PUT, DELETE)

Stateless communication (each request contains all required information)

2. REST is Resource -Based

Unlike action-based APIs (`getSomething()`, `saveSomething()`), REST APIs use **resource-based URIs**:

Resource	URI (Endpoint)
Customers collection	`/customers`
Specific customer	`/customers/1234`
Orders of a customer	`/customers/1234/orders`
Specific order	`/customers/1234/orders/43`

3. **HTTP Methods as Actions**

HTTP methods represent **CRUD** (Create, Read, Update, Delete) operations:

Table 5-1. *CRUD Mapping for HTTP Methods*

Operation	Method	Resource	Example	Remarks
Read – List	GET	Collection	`GET /customers`	Lists objects (additional query string can be used for filtering and sorting)
Read	GET	Object	`GET /customers/1234`	Returns a single object (query string may be used to specify which fields)
Create	POST	Collection	`POST /customers`	Creates an object with the values specified in the body
Update	PUT	Object	`PUT /customers/1234`	Replaces the object with the one specified in the body
Update	PATCH	Object	`PATCH /customers/1234`	Modifies some properties of the object, as specified in the body
Delete	DELETE	Object	`DELETE /customers/1234`	Deletes the object

Altho

REST by itself lays down no rules for the following:

- Filtering, sorting, and paginating a list of objects. The query string is commonly used in an implementation -specific way to specify these.
- Specifying which fields to return in a READ operation.
- If there are embedded objects, specifying which of those to expand in a READ operation.
- Specifying which fields to modify in a PATCH operation.
- Representation of objects. You are free to use JSON, XML, or any other representation for the objects in both READ and WRITE operation

GraphQL

While REST follows a structured API design with multiple endpoints, GraphQL introduces a **single endpoint** that enables clients to request **only the data they need**.

Shortcomings of REST APIs:

1. **Over-fetching**: REST APIs return **more data than necessary** because predefined endpoints deliver full objects.
2. **Under-fetching**: Clients often need to make **multiple API requests** to get related data.
3. **Versioning Issues**: Adding new fields often requires **creating a new API version**.
4. **Multiple Endpoints**: REST requires **separate endpoints for different resources**, making API management more complex.

Key Features of GraphQL

Field Specification (Fine-Grained Data Control)

- In REST, the API response **includes all fields** of an object.
- In GraphQL, the client **must specify** the required fields.
- This reduces **network load** , making it efficient for mobile and Web appliactions

Example GraphQL Query:

```
{ user(id: "1234") {
    name email
  }
}
```

Response (only requested fields are returned):

```
{
  "data": {
    "user": {
      "name": "Alice",
      "email": "alice@example.com"
    }
  }
}
```

Graph-Based Data Model (Natural Relationship Handling)

- REST is **resource-based**, treating each entity as a separate resource.
- GraphQL is **graph-based**, **naturally supporting relationships** between objects.

Example: Fetch a user and their assigned issues in **one** request:

```
{ user(id: "1") { name
    issues { title status
    }
  }
}
```

Single Endpoint (Efficient API Structure)

- **REST APIs** require multiple endpoints (/users, /orders, /products).
- **GraphQL APIs** use a **single endpoint** (e.g., /graphql).
- The request structure (query) defines which data should be returned.

Strongly Typed Schema (Error Prevention & Data Validation)

GraphQL enforces **strong data types** using a schema language.

- Every field and argument has a **defined type**.
- This ensures **valid data queries** and provides **descriptive error messages**. **Example GraphQL**

- **Schema:**

type User { id: ID! name: String! email: String! } **Introspection (Self-Documenting APIs)**

GraphQL servers **can describe their own schema** dynamically.

- Developers can explore available queries and mutations **without external documentation**.
- Tools like **Apollo Playground** allow real-time API exploration and testing.

Libraries

- Parsing and dealing with the type system language (also called the GraphQL Schema Language) as well as the query language is hard to do on your own.
- Fortunately, there are tools and libraries available in most languages for this purpose.
- For JavaScript on the back-end, there is a reference implementation of GraphQL called GraphQL.js. To tie this to Express and enable HTTP requests to be the transport mechanism for the API calls, there is a package called express-graphql.
- But these are very basic tools that lack some advanced support such as modularized schemas and seamless handling of custom scalar types.
- The package graphql-tools and the related apollo-server are built on top of GraphQL.js to add these advanced features. We will be using the advanced packages for the Issue Tracker application in this chapter.
- I will cover only those features of GraphQL that are needed for the purpose of the application.
- For advanced features that you may need in your own specific application, do refer to the complete documentation of GraphQL at https://graphql.org and the tools at https://www.apollographql.com/docs/graphql-tools/.

The About API

1. **Installing Required Packages**

To get started, you need the following **npm** packages:

- **graphql**: The base GraphQL package.
- **apollo-server-express**: Apollo Server middleware for Express.js.

Run the following command to install these: npm install graphql@0 apollo-server-express@2

2. **Defining the GraphQL Schema**

GraphQL uses a schema to define the structure of data and operations.

- **Query** type: Defines read operations.
- **Mutation** type: Defines write (update/create/delete) operations.

Example Schema:

```
const typeDefs = `
type Query {
  about: String! // Read operation
} type Mutation {
}
`; setAboutMessage(message: String!): String // Write operation
```

- about: A **mandatory** (!) string field that provides the about message.
- setAboutMessage(message: String!): A mutation that updates the about message.

3. **Creating Resolvers**

Resolvers are functions that define how each GraphQL field should behave.

Example Resolvers: let aboutMessage = "Issue Tracker API v1.0"; // Initial message

```
const resolvers = {
Query: { about: () => aboutMessage, // Returns the message
},
Mutation: { setAboutMessage(_, { message }) { return
  (aboutMessage = message); // Updates the message
   },
},
};
```

- **Query Resolver**: ○ about: Returns the aboutMessage string.
- **Mutation Resolver**:
  - ○ setAboutMessage: Updates aboutMessage with the new value.

4. **Setting Up Apollo Server**

Apollo Server provides an easy way to integrate GraphQL with Express. const { ApolloServer } = require('apollo-server-express');

```
const server = new ApolloServer({
 typeDefs, resolvers, });
```

This creates a GraphQL server instance using the schema (typeDefs) and resolvers.

5. **Integrating with Express.js**

To run the GraphQL server, we integrate it with an Express application:

```
const express = require('express'); const app = express();

app.use(express.static('public')); // Serve static files

server.applyMiddleware({ app, path: '/graphql' }); // Mount GraphQL endpoint
app.listen(3000, function () {
 console.log('App started on port 3000'); });
```

- **Express** is used to create the HTTP server.
- **GraphQL API is exposed at** /graphql.
- **Server runs on** http://localhost:3000.
-

6. **Using GraphQL Playground**

Apollo Server provides a built-in **GraphQL Playground**, accessible at: http://localhost:3000/graphql This tool allows you to:

- Explore the schema.
- Run queries and mutations.
- View introspection details.

7. **Testing the API**

You can use GraphQL Playground to run the following queries

1.Read the about message:

```
{
query {
  about
}
```

Response:

```
{
  "data": {
    "about": "Issue Tracker API v1.0"
  }
}
```

2.Update the about message:

```
mutation {
  setAboutMessage(message: "New API Version")
}
```

Response:

```
{
  "data": {
                    "setAboutMessage": "New API Version"
  }
}
```

3. Verify the update:

```
query {
  about
}
```

Response:

```
{
}
“data”:{
“about”: “New API Version”}
  }
}
```

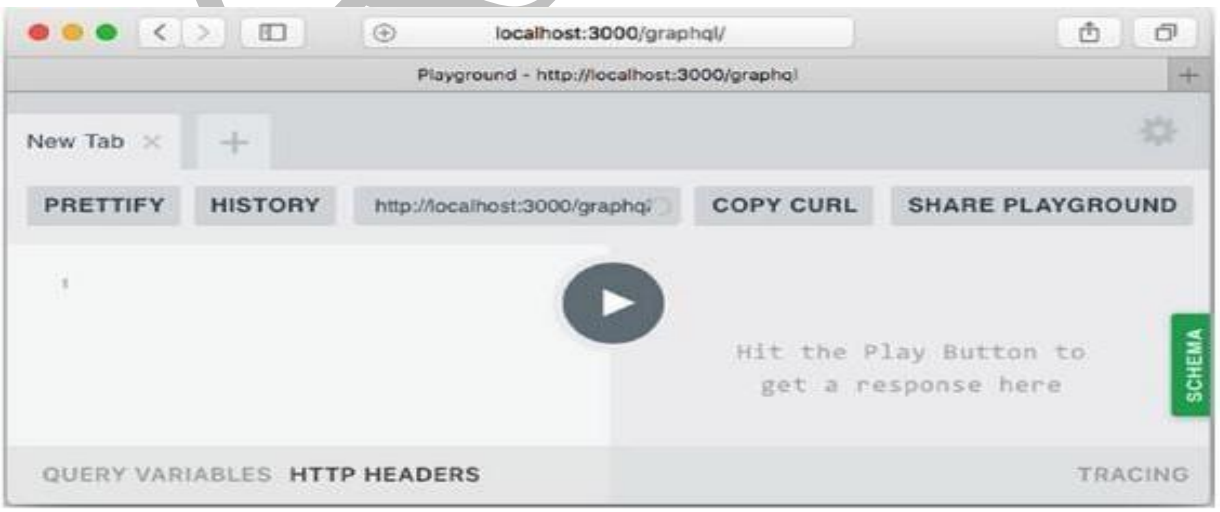

Figure 5-1. *The GraphQL Playground*

GraphQL Schema File

This is a great approach for keeping your GraphQL schema manageable as it grows! Extracting the schema into a separate .graphql file not only makes your code more readable but also improves maintainability.

Key Takeaways:

1. **Separation of Concerns:** Moving the schema to a .graphql file keeps the server.js file clean and modular.
2. **File Reading:** The fs.readFileSync function reads the schema file into a string for ApolloServer.
3. Nodemon Watch: **Since nodemon only watches .js files by default, adding -**e js,graphql ensures it also watches .graphql files.

Potential Enhancements:

- **Use fs.promises.readFile for async operations:**
  Instead of readFileSync, you can use fs.promises.readFile with await for a non-blocking approach:
  const fs = require('fs').promises;
  const typeDefs = await fs.readFile('./server/schema.graphql', 'utf-8');
- **Organizing the Project:**
  - Place your schema in a graphql/ folder (server/graphql/schema.graphql).
  - Structure resolvers in a separate file (server/graphql/resolvers.js).

The List API

1. Define the GraphQL Schema (schema.graphql)

- You introduce a **custom type** Issue that represents an issue object.
- Since GraphQL doesn't have a built-in **Date** type, you use **String** for created and due.
- Add a new **Query field** issueList to return an **array of issues** ([Issue!]!
  ensures a non-nullable array with non-nullable elements).

Now that you have learned the basics of GraphQL, let's make some progress toward building the Issue Tracker application using this knowledge. The next thing we'll do is implement an API to fetch a list of issues. We'll test it using the Playground and, in the next section, we'll change the front-end to integrate with this new API.

Let's start by modifying the schema to define a custom type called Issue. It should contain all the fields of the issue object that we have been using up to now. But since there is no scalar type to denote a date in GraphQL, let's use a string type for the time being. We'll implement custom scalar types later in this chapter. So, the type will have integers and strings, some of which are optional. Here's the partial schema code for the new type.

```
...
type Issue {
  id: Int!
  ...
  due: String
}
...
```

Now, let's add a new field under Query to return a list of issues. The GraphQL way to specify a list of another type is to enclose it within square brackets. We could use [Issue] as the type for the field, which we will call issueList.

But we need to say not only that the return value is mandatory, but also that each element in the list cannot be null. So, we have to add the exclamation mark after Issue as well as after the array type, as in [Issue!]!.

Let's also separate the top-level Query and Mutation definitions from the custom types using a comment. The way to add comments in the schema is using the # character at the beginning of a line. All these changes are listed in Listing 5-5

Listing 5-5. schema.graphql: Changes to Include Field issueList and New Issue Type

```
type Issue {
  id: Int!
  title: String!
  status: String!
  owner: String
  effort: Int
  created: String!
  due: String
}

##### Top level declarations

type Query {
  about: String!
  issueList: [Issue!]!
}

type Mutation {
  setAboutMessage(message: String!): String
}
```

Listing 5-6. server.js: Changes for issueList Query Field

```
...
let aboutMessage = "Issue Tracker API v1.0";

const issuesDB = [
  {
    id: 1, status: 'New', owner: 'Ravan', effort: 5,
    created: new Date('2019-01-15'), due: undefined,
    title: 'Error in console when clicking Add',
  },
  {
    id: 2, status: 'Assigned', owner: 'Eddie', effort: 14,
    created: new Date('2019-01-16'), due: new Date('2019-02-01'),
    title: 'Missing bottom border on panel',
  },
];

const resolvers = {
  Query: {
    about: () => aboutMessage,
    issueList,
  },
  Mutation: {
    setAboutMessage,
  },
};

function setAboutMessage(_, { message }) {
  return aboutMessage = message;
}

function issueList() {
  return issuesDB;
}
...
```

To test this in the Playground, you will need to run a query that specifies the issueList field, with subfields. But first, a refresh of the browser is needed so that the Playground has the latest schema and doesn't show errors when you type the query.

```
query {
  issueList {
    id
    title
    created
  }
}
```

This query will result in an output like this.

```
{
  "data": {
    "issueList": [
      {
        "id": 1,
        "title": "Error in console when clicking Add",
        "created": "Tue Jan 15 2019 05:30:00 GMT+0530 (India Standard Time)"
      },
      {
        "id": 2,
        "title": "Missing bottom border on panel",
        "created": "Wed Jan 16 2019 05:30:00 GMT+0530 (India Standard Time)"
      }
    ]
  }
}
```

.

List API Integration

Now that the **GraphQL List API** is functional, we need to **integrate it into the UI** by modifying the IssueList component in React.

1. Include Fetch Polyfill (For Older Browsers)

Since we are using fetch(), we include a **polyfill** for Internet Explorer and older browsers in index.html:

<script src="https://unpkg.com/@babel/polyfill@7/dist/polyfill.min.js"></script>

<script src="https://unpkg.com/whatwg-fetch@3.0.0/dist/fetch.umd.js"></script>

2. Modify the loadData() Method in IssueList Construct GraphQL Query

- Create a GraphQL query string to fetch all issue fields:

```
const query = `query { issueList {
    id title status owner created effort due
  }
}`;
```

Make an API Request using fetch()

- Use the fetch() method to send a **POST** request to /graphql:

```
const response = await fetch('/graphql', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ query })
});
```

Handle the API Response

- Convert the response **JSON** into a JavaScript object and update state:

```
const result = await response.json(); this.setState({ issues:
result.data.issueList });
```

3. Fixing Date Handling in IssueRow Component

- Initially, created and due were Date objects, but now they are **strings**. **Fix:** Remove .toDateString() and use the string values directly:

```
<td>{issue.created}</td>
<td>{issue.due}</td>
```

4. Remove the Hardcoded Initial Issues

- Previously, the initialIssues array was used as default data.
- Since data is now fetched dynamically, **remove initialIssues** from App.jsx.

5. Testing the Integration

- **Refresh the browser**: The issue list will now be loaded from the API.
- The UI **remains the same**, except:
  - **Long date strings** appear instead of formatted dates.
  - **Add operation does not work** (we will fix this in the next section).

Custom Scalar Types

Using Custom Scalar Types for Date in GraphQL

Storing dates as **strings** in the database or API responses **is not ideal** because:

1. **Sorting and filtering become complex** (since string-based sorting does not work correctly for dates).
2. **Time zone and localization issues** (dates should be displayed in the user's local time).
3. **Standardization issues** (different formats can cause inconsistencies).

1. Define a Custom GraphQL Scalar Type for Date

GraphQL does not support Date natively, so we create a **custom scalar type**.

Modify schema.graphql

- Use scalar instead of type to define a **new GraphQL type** for dates. Update the Issue type to use GraphQLDate instead of String. scalar GraphQLDate

```
type Issue { id: Int! title: String! status:
  String! owner: String effort: Int created:
  GraphQLDate! due: GraphQLDate
}
```

2. Implement the Resolver for GraphQLDate

A **GraphQL scalar type resolver** handles:

- **Serialization** (converting a Date to an ISO string for API responses).
- **Parsing** (converting an ISO string back into a Date object when receiving input).

Modify server.js

1. **Import the required package**:

```
const { GraphQLScalarType } = require('graphql');
```

2. **Create the custom GraphQLDate type resolver**:

```
const GraphQLDate = new GraphQLScalarType({ name: 'GraphQLDate',
  description: 'A Date() type in GraphQL as a scalar', serialize(value) {
    return value.toISOString(); // Convert Date to ISO 8601 string
  },
  parseValue(value) {
    return new Date(value); // Convert ISO string to Date object
  },
  parseLiteral(ast) {
    return new Date(ast.value); // Convert AST literal to Date object }
});
```

3. **Include GraphQLDate in the resolvers object**:

```
const resolvers = {
  Query: { about: () => aboutMessage,
    issueList,
  },
  Mutation: { setAboutMessage,
  },
  GraphQLDate, // Add custom scalar type
};
```

3. How This Works

API Response**: When fetching issues, GraphQLDate will convert Date objects to** ISO 8601 strings**.**
API Input Handling**: If a new issue is added with a date in ISO format, it will be** converted back to a Date object **automatically.**

4. Next Steps

1. **Update the front -end** to properly handle GraphQLDate values.
2. **Ensure UI displays dates in a localized format** using toLocaleDateString() or similar methods.
3. **Modify the Add Issue mutation** to accept ISO date strings.

Finally, we need to set this resolver at the same level as Query and Mutation (at the top level) as the value for the scalar type GraphQLDate. The complete set of changes in server.js is shown in Listing 5 -10.

Listing 5-10. server.js: Changes for Adding a Resolver for GraphQLDate

```
...
const { ApolloServer } = require('apollo-server-express');
const { GraphQLScalarType } = require('graphql');
...

const GraphQLDate = new GraphQLScalarType({
  name: 'GraphQLDate',
  description: 'A Date() type in GraphQL as a scalar',
  serialize(value) {
    return value.toISOString();
  },
});

const resolvers = {
  Query: {
    ...
  },
```

```
  Mutation: {
    ...
  },
  GraphQLDate,
};
...
```

At this point, if you switch to the Playground and refresh the browser (due to schema changes), and then test the List API. You will see that dates are being returned as the ISO string equivalents rather than the locale-specific long string previously used. Here's a query for testing in the Playground:

```
query {
  issueList {
    title
    created
    due
  }
}
```

Here are the results for this query:

```
{
  "data": {
    "issueList": [
      {
        "title": "Error in console when clicking Add",
        "created": "2019-01-15T00:00:00.000Z",
        "due": null
      },
      {
        "title": "Missing bottom border on panel",
        "created": "2019-01-16T00:00:00.000Z",
        "due": "2019-02-01T00:00:00.000Z"
      }
    ]
  }
}
```

The Create API

Implementing an API for Creating a New Issue in GraphQL

Now, we will add an API endpoint to **create new issues** in our in-memory database. This involves:

1. **Defining an IssueInputs input type** in schema.graphql.
2. **Adding the issueAdd mutation** to the GraphQL schema.
3. **Implementing the resolver** for issueAdd in server.js.
4. **Updating the GraphQLDate scalar** to handle parsing input dates.

1. Modify schema.graphql

First, define an **input type** IssueInputs (separate from the Issue type because it does not include id or created).

"Toned down Issue, used as inputs, without server-generated values." input IssueInputs {
  title: String!
  "Optional, if not supplied, will be set to 'New'" status: String
  owner: String effort: Int
  due: GraphQLDate
}
type Mutation { setAboutMessage(message: String!): String
  issueAdd(issue: IssueInputs!): Issue!

}

- **Mandatory fields**: title (required with !).
- **Optional fields**: status, owner, effort, due (GraphQLDate). □ **Default value**: status is set to "New" if not provided.

2. Implement issueAdd Resolver in server.js

Now, implement the resolver function to:

- **Generate id and created fields automatically**.
- **Default status to 'New' if missing**. □ **Push the issue to issuesDB**.

Modify server.js

1. **Define issueAdd function**:

```
const issuesDB = []; // In-memory storage

function issueAdd(_, { issue }) {
  issue.created = new Date();
  issue.id = issuesDB.length + 1;
  if (issue.status === undefined) {
    issue.status = 'New';
  }
  issuesDB.push(issue); return issue;
}
```

2. **Add issueAdd to the Mutation resolver**:

```
const resolvers = {
  Query: { about: () => aboutMessage,
    issueList,
  },
  Mutation: { setAboutMessage, issueAdd, //
    New Mutation
  },
  GraphQLDate, // Custom Date scalar
};
```

3. Update GraphQLDate Resolver to Handle Parsing

Since IssueInputs includes GraphQLDate, we need to update the **date scalar resolver**.

1. **Import Kind from graphql/language**:

```
const { GraphQLScalarType, Kind } = require('graphql');
```

2. **Modify GraphQLDate resolver**:

```
const GraphQLDate = new GraphQLScalarType({ name: 'GraphQLDate',
  description: 'A Date() type in GraphQL as a scalar', serialize(value) {
    return value.toISOString(); // Convert Date to ISO 8601 string
  },
  parseValue(value) {
    return new Date(value); // Convert ISO string to Date object
  },
  parseLiteral(ast) {
    return ast.kind === Kind.STRING ? new Date(ast.value) : undefined; }
});
```

4. Testing the issueAdd Mutation

Now, test the API by sending the following mutation request in GraphiQL or Postman:

```
mutation {
  issueAdd(issue:  {  title:  "GraphQL
    Issue", owner: "Alice",
    effort: 5,
    due: "2025-02-15T12:00:00.000Z"
  }) { id title status
    owner
    effort created due
  }
}
```

Expected Response

```
{
  "data": {
    "issueAdd": {
      "id": 1,
      "title": "GraphQL Issue",
      "status": "New",
      "owner": "Alice",
      "effort": 5,
      "created": "2025-02-12T10:30:00.000Z",
      "due": "2025-02-15T12:00:00.000Z"
    }
  }
}
```

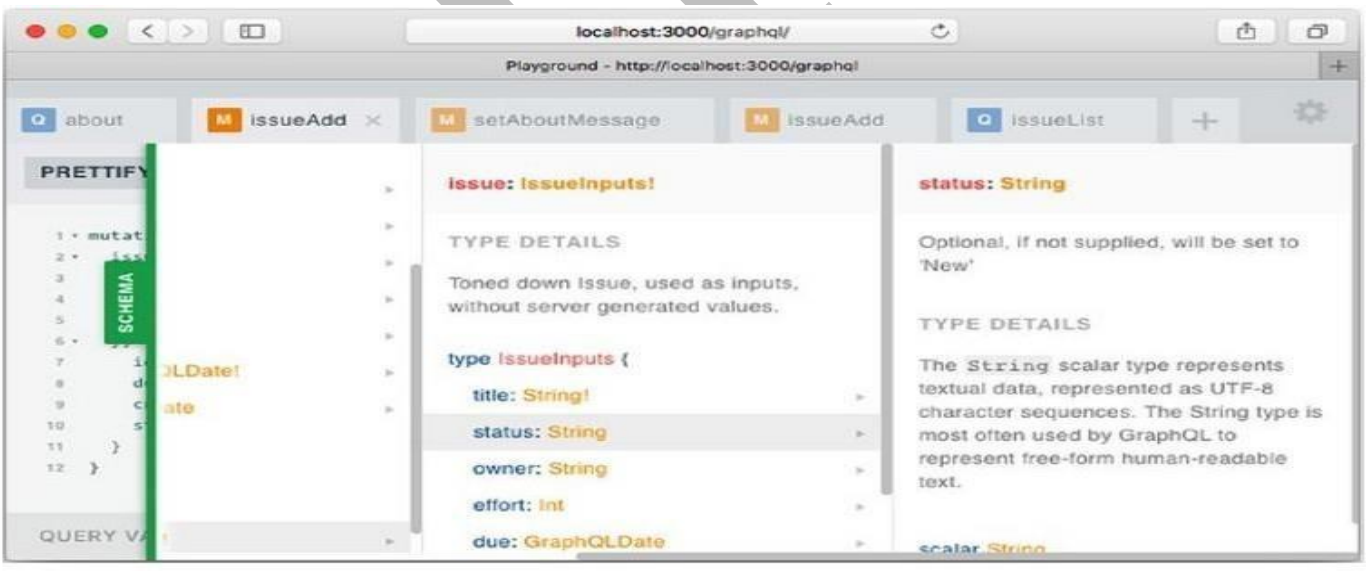

Figure 5-3. *Schema showing descriptions of IssueInputs and status*

CreateAPI Integration

In this integration, we modify the **IssueAdd** component and the **createIssue** function to send a new issue to the server via GraphQL mutation.

1. Modify IssueAdd to Handle User Input

- We remove the default "status": "New" from the frontend since the backend handles it.
- We set the due date to **10 days from today** using JavaScript.

```
const issue = {
  owner: form.owner.value, title: form.title.value,
  due: new Date(new Date().getTime() + 1000 * 60 * 60 * 24 * 10), // 10 days later
};
```

2. Construct the GraphQL Mutation Query

- We create a mutation query using a **template string**.
- Since GraphQL requires date fields as strings, we convert due to **ISO format**.
- We only request the id field in the response.

```
const query = `mutation { issueAdd(issue:{
    title:  "${issue.title}",  owner:  "${issue.owner}",
    due: "${issue.due.toISOString()}"
  }) { id
  }
}`;
```

3. Send the GraphQL Request Using fetch

We send query using a POST request with JSON body.
Headers specify that the request conatin JSON

```
const response = await fetch('/graphql', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ query })
});
```

4. Refresh the Issue List Instead of Updating State Manually

- Instead of adding the new issue to state.issues, we **reload the entire issue list**.
- This ensures data accuracy in case of errors or concurrent updates.

```
this.loadData(); // Reload issue list
```

5. Result

- The new issue is added with a **due date of 10 days from now**.
- Refreshing the page shows the new issue because it is stored on the server.

Query Variables

Instead of embedding dynamic values directly inside the query string, GraphQL allows us to use **query variables**, which are passed separately in a JSON object. This approach improves **security, readability, and reliability** by avoiding issues with escaping special characters.

Why Use Query Variables?

1. **Avoids string concatenation issues** – Prevents errors with special characters like quotes (") and curly braces ({}).
2. **More secure** – Similar to prepared statements in SQL, reducing risks like **GraphQL Injection**.
3. **Easier debugging** – Queries remain **clean** while variables are passed separately.

How to Use Query Variables in a Mutation?

1. Name the Mutation

We name the mutation (setNewMessage) and replace **static values** with variables (starting with $).

Before (hardcoded value inside query):

```
mutation {
  setAboutMessage(message: "New About Message")
}
```

After (using a variable $message):

```
mutation setNewMessage($message: String!) {
  setAboutMessage(message: $message)
}
```

2. Declare Variables in JSON Format

When executing this query (e.g., in GraphQL Playground or an API call), the **variables must be passed separately** as JSON.

```
{
  "message": "Hello World!"
}
```

Applying This to issueAdd Mutation

Instead of inserting values directly in a template string, we pass them as variables.

Before (string template approach, error-prone)

```
const query = `mutation {
  issueAdd(issue:{
    title: "${issue.title}", owner: "${issue.owner}",
    due: "${issue.due.toISOString()}"
  }) { id
  }
}`;
```

After (query variables approach, safer and cleaner)

```
const query = `mutation
addIssue($issue: IssueInputs!) {
  issueAdd(issue: $issue) { id
  } }`;
const variables = { issue: {
    title: issue.title,
    owner: issue.owner,
    due: issue.due.toISOString()
  }
};
const response = await fetch('/graphql', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' }, body: JSON.stringify({
  query, variables })
});
```

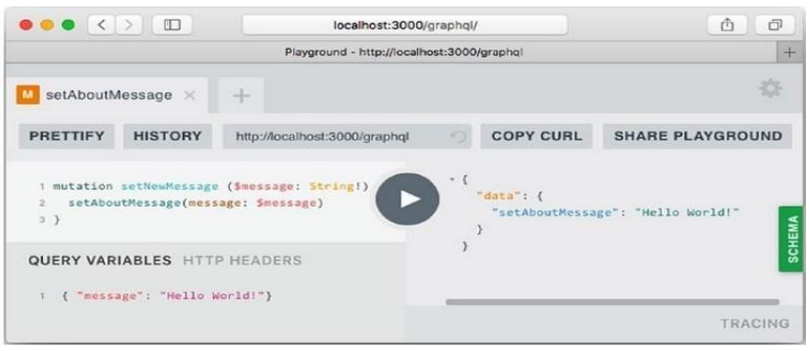

Figure 5-4. Playground with query variables

Benefits of Using Query Variables

✔**More Secure** – Avoids issues with special characters in user input.

✔**Easier Debugging** – Query stays static while variables are dynamic.

✔**Improves Performance** – GraphQL servers can **cache and optimize** queries better.

Input Validations

1. Input Validations :Input validation ensures that user input meets specific criteria before being processed by the server. In GraphQL, we can implement validation in two primary ways:

a. Schema-Level Validations

- **Using Enums for Restricted Values:**
  - Instead of using a plain string for certain fields (e.g., status), GraphQL allows us to use enums to limit the possible values. ○ Example:

```
enum StatusType {
  New
  Assigned
  Fixed
  Closed }
type Issue {
  status: StatusType! }
```

  - If a user provides an invalid value (e.g., "Unknown" instead of New), GraphQL will return a validation error automatically.
- **Providing Default Values:**
  - Default values can be assigned to fields to prevent missing values.
  - Example:

```
input IssueInputs {
  status: StatusType = New }
```

  - If a user does not provide a status, it will default to "New".

b. Programmatic Validations

- Implemented in the resolver function before saving data.
- Example validation rules:
  - **Title must be at least 3 characters long ○ Owner is required when status is "Assigned" ○ Invalid date values should be detected**

Example:

```
function validateIssue(_, { issue }) {
  const errors = [];
  if (issue.title.length < 3) {
                errors.push('Field "title" must be at least 3 characters long.');
  }
  if (issue.status === 'Assigned' && !issue.owner) {
    errors.push('Field "owner" is required when status is "Assigned"'); }
if (errors.length > 0) {
    throw new UserInputError('Invalid input(s)', { errors }); }
}
```

- **Validating Dates:**

Example: Ensuring the provided date is in the correct format

```
parseValue(value){const dateValue = new Date(value);
return isNaN(dateValue) ? undefined : dateValue;}
```

Displaying Errors

Errors must be displayed properly in the UI to improve the user experience. We handle two types of errors:

a. Transport Errors (e.g., Network Issues)

- These errors occur when there is a problem with the API request.
- Handled using a try-catch block around the fetch function.
- Example:

```
async function graphQLFetch(query, variables = {}) {
  try {
    const response = await fetch('/graphql', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' }, body: JSON.stringify({
      query, variables })
    });
const result = await response.json(); return result.data;
  } catch (e) { alert(`Error in sending data to server: ${e.message}`); }
}
```

b. User Input Errors (Validation Failures)

- These errors occur when the user provides invalid data (e.g., empty title, missing owner).
- Displaying errors for user input:

```
if (result.errors) {
  const error = result.errors[0];
if (error.extensions.code === 'BAD_USER_INPUT') {
    const details = error.extensions.exception.errors.join('\n '); alert(`${error.message}:\n ${details}`);
  } else { alert(`${error.extensions.code}: ${error.message}`); }
}
```

c. Example Error Messages

1. **Missing Title**

```
{ "message": "Invalid input(s)",
  "errors": ["Field \"title\" must be at least 3 characters long."] }
```

2. **Invalid Date**

```
{ "message": "Expected type GraphQLDate, found \"not-a-date\"." }
```

3. **Invalid Status**

```
{ "message": "Expected type StatusType, found \"Unknown\"." }
```

To test transport errors, you can stop the server after refreshing the browser and then try to add a new issue. If you do that, you will find the error message like the screenshot in Figure 5-5.

Figure 5-5. Transport error message

MongoDB: Basics, Documents, Collections, Databases, Query Language, Installation, The Mongo Shell, MongoDB CRUD Operations, Create, Read, Projection, Update, Delete, Aggregate, MongoDB Node.js Driver, Schema Initialization, Reading from MongoDB, Writing to MongoDB.

Text Book 2: Chapter 6, 7

5 Module

Module -5 Lecturer Notes: – MongoDB: Basics

This Lecture Notes provides a comprehensive overview of **Shared-memory programming with OpenMP**, based on Chapter 5 of "Peter S Pacheco, Matthew Malensek – **MongoDB**: Basics, Documents, Collections, Databases, Query Language, Installation, The Mongo Shell, MongoDB CRUD Operations, Create, Read, Projection, Update, Delete, Aggregate, MongoDB Node.js Driver, Schema Initialization, Reading from MongoDB, Writing to MongoDB.

MODULE-5: MangoDB

1. **MongoDB**: Basics,
2. Documents,
3. Collections,
4. Databases, Query Language,
5. Installation,
6. The Mongo Shell,
7. MongoDB CRUD Operations,
8. Create,
9. Read,
10. Projection,
11. Update,
12. Delete,
13. Aggregate,
14. MongoDB Node.js Driver,
15. Schema Initialization,
16. Reading from MongoDB,
17. Writing to MongoDB.

Text Book 2: Chapter 6, 7

MangoDB Basics

MangoDB is a NoSQL database that stores data in a flexible, JSON-like format called document**s** instead of tables. It is designed for **sc**alability**,** high performance, and ease of development.

Core Concepts:

Documents:

The basic unit of data storage in MongoDB, similar to a row in relational databases but in JSON format.

MongoDB is a document-oriented database, meaning that data is stored in document**s** rather than tables. These documents are structured as field-value pairs, similar to JSON objects, allowing for flexibility and scalability.

MongoDB Document Structure:A document in MongoDB consists of fields (keys) and values, where values can be:

- **Primitive types** (strings, numbers, Booleans, dates, timestamps, binary data, etc.)
- **Embedded objects** (nested key-value pairs)
- **Arrays** (lists of values or objects)

For example, an **Invoice document** in MongoDB might look like this:

```
{
  "invoiceNumber": 1234,
  "invoiceDate": ISODate("2018-10-12T05:17:15.737Z"),
  "billingAddress": {
    "name": "Acme Inc.",
    "line1": "106 High Street",
    "city": "New York City",
    "zip": "110001-1234"
  },
  "items": [
    {
      "description": "Compact Fluorescent Lamp",
      "quantity": 4,
      "price": 12.48
    },
    {
      "description": "Whiteboard",
      "quantity": 1,
      "price": 5.44
    }
  ]
}
```

Explanation of the Document

1. **Basic Fields** ○ "invoiceNumber": 1234 → Stores an invoice number as a number.
    - "invoiceDate": ISODate("2018-10-12T05:17:15.737Z") → Uses MongoDB's date format for timestamps.
2. **Nested Object (billingAddress)** ○ Instead of storing address details in a separate table (like in SQL), MongoDB stores it **inside the document**.
    - "billingAddress" contains fields like name, line1, city, and zip.
3. **Array of Objects (items)** ○ Instead of using a separate table for invoice items, MongoDB allows an **array of objects** within the document. ○ Each item contains "description", "quantity", and "price".

Advantages of This Approach in MongoDB

Faster Reads – No need for complex joins; all data is retrieved in a single query. **Flexible Schema** – No need to define a fixed structure; fields can be added or modified easily.**Efficient Storage** – Stores related data together, reducing redundancy.

Better Scalability – Easily scales horizontally across multiple servers.

Collections:

A group of related documents, similar to tables in SQL databases.

MongoDB Collections and Schema

In MongoDB, a collection is similar to a table in a relational database—it is a group of related documents. However, MongoDB collections offer more flexibility than relational tables.

Key Features of MongoDB Collections

1. **Primary Key (_id Field)** ○ Every document must have a unique _id field. ○ If not provided, MongoDB automatically generates a unique ObjectId for _id.
    - The _id field is automatically indexed for fast lookups.
2. **Indexes for Faster Queries** ○ Apart from _id, indexes can be created on other fields to improve query performance.
    - Indexes can also be created for embedded documents and arrays.
3. **Schema Flexibility** ○ Unlike relational databases, MongoDB does not require a predefined schema.
    - Documents within the same collection can have different fields.
    - However, in practice, most applications follow a consistent document structure.
4. **Schema Validation (Optional, Introduced in MongoDB 3.6)** ○ Allows defining required fields, data types, string lengths, and value rang**es**.
    - Ensures data integrity but is **o**ptional.
    - Errors from schema validation are currently not very detailed.

Example: A MongoDB Collection (users)

```
{
  "_id": ObjectId("65a4e6b8c7e9a8d5b1d9a456"),
  "name": "Alice",
  "email": "alice@example.com",
  "age": 30,
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "zip": "10001"
  }
}
```

Here,

- Id is automatically generated if not provided.
- The document has a nested object (address)**,** showing flexibility.
- If another document in the same collection lacks address, it is still valid.

Databases

- A database is a logical grouping of many collections. Since there are no foreign keys like in a SQL database, the concept of a database is nothing but a logical partitioning namespace.

- Most database operations read or write from a single collection, but $lookup, which is a stage in an aggregation pipeline, is equivalent to a join in SQL databases.
- This stage can combine documents within the same database. Further, taking backups and other administrative tasks work on the database as a unit.
- A database connection is restricted to accessing only one database, so to access multiple databases, multiple connections are required.
- Thus, it is useful to keep all the collections of an application in one database, though a database server can host multiple databases

Query Language

- Unlike the universal English-like SQL in a relational database, the MongoDB query language is made up of methods to achieve various operations.
- The main methods for read and write operations are the CRUD methods.
  Other methods include aggregation, text search, and geospatial queries.
- The query filter is a JavaScript object consisting of zero or more properties, where the property name is the name of the field to match on and the property value consists of another object with an operator and a value.
- For example, to match all documents with the field invoiceNumber that are greater than 1,000, the following query filter can be used:

  { "invoiceNumber": { $gt: 1000 } }

- Since there is no "language" for querying or updating, the query filters can be very easily constructed programmatically.
- Unlike relational databases, MongoDB encourages denormalization, that is, storing related parts of a document as embedded subdocuments rather than as separate collections (tables) in a relational database.
- Take an example of people (name, gender, etc.) and their contact information (primary address, secondary address etc.).
- In a relational database, this would require separate tables for People and Contacts, and then a join on the two tables when all of the information is needed together.
- In MongoDB, on the other hand, it can be stored as a list of contacts within the same People document. That's because a join of collections is not natural to most methods in MongoDB: the most convenient find() method can operate only on one collection at a time.

Installation

Before you try to install MongoDB on your computer, you may want to try out one of the hosted services that give you access to MongoDB. There are many services, but the following are popular and have a free version that you can use for a small test or sandbox application. Any of these will do quite well for the purpose of the Issue Tracker application that we'll build as part of this book.

- MongoDB Atlas (https://www.mongodb.com/cloud/atlas): I refer to this as Atlas for short. A small database (shared RAM, 512 MB storage) is available for free.
- mLab (previously MongoLab) (https://mlab.com/): mLab has announced an

acquisition by MongoDB Inc. and may eventually be merged into Atlas itself. A sandbox environment is available for free, limited to 500 MB storage.

- Compose (https://www.compose.com): Among many other services, Compose offers MongoDB as a service. A 30-day trial period is available, but a permanently free sandbox kind of option is not available.

On a Windows system, you may need to append .exe to the command. The command may require a path depending on your installation method. If the shell starts successfully, it will also connect to the local MongoDB server instance. You should see the version of MangoDB printed on the console, the database it is connecting to (the default is test), and a command prompt, like this, if you had installed MongoDB version 4.0.2 locally:

```
MongoDB shell version v4.0.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 4.0.2
>
```

The Mongo Shell

The mongo shell is an interactive JavaScript shell, very much like the Node.js shell. In the interactive shell, a few non-JavaScript conveniences are available over and above the full power of JavaScript. In this section, we'll discuss the basic operations that are possible via the shell, those that are most commonly used. For a full reference of all the capabilities of the shell, you can take a look at the mongo shell documentation at https://docs.mongodb.com/manual/mongo/.

This will list the databases and the storage occupied by them. For example, in a fresh local installation of MongoDB, this is what you will see:

```
admin          0.000GB
config         0.000GB
local          0.000GB
```

These are system databases that MongoDB uses for its internal book keeping, etc. We will not be using any of these to create our collections, so we'd better change the current database. To identify the current database, the command is:

> db

The default database a mongo shell connects to is called test and that is what you are likely to see as the output to this command. Let's now see what collections exist in this database.

> show collections

You will find that there are no collections in this database, since it is a fresh installation. Further, you will also find that the database test was not listed when we listed the available databases. That's because

databases and collections are really created only on the first write operation to any of these. Let's switch to a database called issuetracker instead of using the default database:

> use issuetracker

This should result in output that confirms that the new database is issuetracker:

```
switched to db issuetracker
```

Let's confirm that there are no collections in this database either:

> show collections

This command should return nothing. Now, let's create a new collection. This is done by creating one document in a collection.

Apart from the insertOne() method, many methods are available on any collection. You can see the list of available methods by pressing the Tab character twice after typing "db.employees." (the period at the end is required before pressing Tab). You may find an output like the following:

CRUD

```
db.employees.addIdIfNeeded(              db.employees.getWriteConcern(
db.employees.aggregate(                  db.employees.group(
db.employees.bulkWrite(                  db.employees.groupcmd(
db.employees.constructor                 db.employees.hasOwnProperty
db.employees.convertToCapped(            db.employees.hashAllDocs(
db.employees.convertToSingleObject(      db.employees.help(
db.employees.copyTo(                     db.employees.initializeOrderedBulkOp(
db.employees.count(                      db.employees.initializeUnorderedBulkOp(
db.employees.createIndex(                db.employees.insert(
db.employees.createIndexes(              db.employees.insertMany(
db.employees.dataSize(                   db.employees.insertOne(
...
```

Thi i t aut -completion feature o t mongo shel a work Not th yo ca let the mongo shell auto -complete the name of any method by pressing the Tab character after entering the beginning few characters of the method.

MongoDB CRUD Operations

operations in MongoDB refer to the basic database operations:

1. **Create** – Insert new documents into a collection.
    - insertOne(): Inserts a single document.

- insertMany(): Inserts multiple documents.

```
db.employees.insertOne({ id: 1, name: { first: 'John', last: 'Doe' }, age: 44 });
db.employees.insertMany([{ id: 2, name: { first: 'Jane', last: 'Doe' }, age: 30 }]);
```

2. **Read** – Retrieve documents from a collection.
   - findOne(): Fetches a single document. ○ find(): Fetches multiple documents with filtering options.

```
db.employees.findOne({ id: 1 });
db.employees.find({ age: { $gte: 30 } });
```

3. **Update** – Modify existing documents.
   - updateOne(): Updates the first matching document. ○ updateMany(): Updates multiple documents.
   - $set: Used to modify specific fields. ○ replaceOne(): Replaces an entire document.

```
db.employees.updateOne({ id: 1 }, { $set: { age: 45 } });
db.employees.updateMany({}, { $set: { organization: 'MyCompany' } });
```

4. **Delete** – Remove documents from a collection. ○ deleteOne(): Deletes a single document.
   - deleteMany(): Deletes multiple documents.

```
db.employees.deleteOne({ id: 4 }); db.employees.deleteMany({ age: { $lt: 20 } });
```

Aggregate

Aggregation in MongoDB is a process of transforming and analyzing data by applying operations such as filtering, grouping, and computations. It allows us to summarize and manipulate data similar to SQL's GROUP BY but with additional capabilities such as joins ($lookup), array expansion ($unwind), and complex transformations.

MongoDB provides the Aggregation Framework, which processes documents through a sequence of pipeline stages. Each stage modifies the data before passing it to the next.

Basic Aggregation Example

The find() method retrieves raw documents, but aggregate() allows for summarization.

1. Summing a Field

To calculate the total age of all employees:

```
db.employees.aggregate([
  { $group: { _id: null, total_age: { $sum: '$age' } } }
])
```

Output:

```
{ "_id" : null, "total_age" : 103 }
```

2. Counting Documents To count the number of

employees:

```
db.employees.aggregate([
  { $group: { _id: null, count: { $sum: 1 } } }
])
```

Output:

```
{ "_id" : null, "count" : 3 }
```

3. Grouping Data by a Field

Let's group employees by their organization and calculate the total age for each: db.employees.aggregate([

])

{ $group: { _i '$organization', total_age: { $sum: '$age } } }

Output:

```
{ "_id" : "OtherCompany", "total_age" : 64 }
{ "_id" : "MyCompany", "total_age" : 103 }
```

4. Calculating Average

To compute the average age of employees per organization:

```
db.employees.aggregate([
  { $group: { _id: '$organization', average_age: { $avg: '$age' } } }
])
```

Output:

```
{ "_id" : "OtherCompany", "average_age" : 64 }
{ "_id" : "MyCompany", "average_age" : 34.33 }
```

Key Features of Aggregation

1. **Pipeline Processing** – Stages process data step-by-step.
2. **Grouping & Summarization** – $group, $sum, $avg, $min, and $max perform statistical operations.
3. **Filtering & Sorting** – $match filters documents, $sort orders them.
4. **Transforming Data** – $project modifies output fields.
5. **Array Processing** – $unwind expands arrays into separate documents.
6. **Joins** – $lookup enables data fetching from other collections.

MongoDB Node.js Driver

The MongoDB Node.js driver is a library that allows applications built with Node.js to connect to a MongoDB database, interact with collections, and perform CRUD operations. It is similar to using the MongoDB shell but with an API that integrates into JavaScript and Node.js applications.

MongoDB provides a low-level driver as well as an Object Document Mapper (ODM) like Mongoose. The low-level driver offers direct control over the database operations, making it a good choice for understanding how MongoDB works at its core. However, ODMs like Mongoose add a layer of abstraction and provide a more structured approach.

1. Installing the MongoDB Node.js Driver

To use the MongoDB driver, first, install it in your Node.js application:

npm install mongodb@3

2. Connecting to a MongoDB Server

To connect to MongoDB, you need to use the MongoClient object. The connection URL specifies the server address and the database.

Basic Connection Code const { MongoClient } = require('mongodb'); const url = 'mongodb://localhost/issuetracker'; // Local MongoDB URL const client = new MongoClient(url, { useNewUrlParser: true });

client.connect((err, client) => { if (err) { console.error("Error connecting to MongoDB:", err); return;
}
console.log('Connected to MongoDB');
const db = client.db(); // Get the database instance client.close(); // Close the connection after operations });

Connection URL Format

- Local MongoDB:
  mongodb://localhost/issuetracker
- Cloud-based

MongoDB Atlas:

mongodb+srv://UUU:PPP@cluster0-XXX.mongodb.net/issuetracker?retryWrites=true

  - Replace UUU with the username, PPP with the password, and XXX with the cluster hostname.

3. Performing CRUD Operations

Once connected, we can interact with collections using the database object (db).

3.1 Inserting a Document

The insertOne() method is used to add a document to a collection. This method is asynchronous and requires a callback function or async/await.

const collection = db.collection('employees'); // Get the collection const employee = { id: 1, name: 'A. Callback', age: 23 };
collection.insertOne(employee, (err, result) => { if (err) {
console.error("Error inserting document:", err); return; }
console.log('Inserted Document ID:', result.insertedId);
});

3.2 Querying (Finding) Documents

if (err) {

To retrieve documents, use find() followed by .toArray().

collection.find({ _i result.insertedId } .toArray((err, docs = {

console.error("Error retrieving documents:", err);
return; }
console.log('Found Documents:', docs);
});

3.3 Updating a Document

Use updateOne() to modify a document.

```
collection.updateOne({ id: 1 }, { $set: { age: 30 } }, (err, result) => { if (err) {
    console.error("Error updating document:", err); return; }
    console.log('Modified Count:', result.modifiedCount);
});
```

3.4 Deleting a Document

To delete a document, use deleteOne().

```
collection.deleteOne({ id: 1 }, (err, result) => { if (err) {
    console.error("Error deleting document:", err); return; }
    console.log('Deleted Count:', result.deletedCount);
});
```

4. Using Async/Await Instead of Callbacks

The callback-based approach can lead to "callback hell" due to deeply nested functions. A cleaner approach is using async/await.

Refactored Code with Async/Await async function testWithAsync() { console.log('\n--- testWithAsync ---'); const client = new MongoClient(url, { useNewUrlParser: true });

```
    try { await client.connect(); // Wait for the connection to establish
        console.log('Connected to MongoDB');
        const db = client.db(); const collection =
        db.collection('employees');
        const employee = { id: 2, name: 'B. Async', age: 16 };

        const result = await collection.insertOne(employee); console.log('Inserted Document ID:',
        result.insertedId);

        const docs = await collection.find({ _id: result.insertedId }).toArray(); console.log('Found
        Documents:', docs);
    } catch (err) { console.error("Error:", err);
    } finally { client.close();
    }
}
 // Call the function testWithAsync();
```

Why Use Async/Await?

- Cleaner and more readable code.
- Avoids deeply nested callbacks.
- Uses try...catch for better error handling.

5. Running the MongoDB Script

To run the script, save it as trymongo.js and execute: node scripts/trymongo.js

Before running, clear the collection to prevent duplicate key errors:

```
mongo issuetracker --eval "db.employees.remove({})"
```

For MongoDB Atlas:

```
mongo "mongodb+srv://cluster0-xxxxx.mongodb.net/issuetracker" --username atlasUser --password
atlasPassword --eval "db.employees.remove({})"
```

6. Error Handling

Errors in MongoDB operations should be handled properly using:

1. **Callback-based error handling**

```
if (err) { console.error("Error:", err);
    client.close(); return;
}
```

2. **Using Try-Catch in Async/Await**

```
try { await client.connect();
} catch (err) { console.error("Connection error:", err); }
```

7. Indexing and Unique Constraints

MongoDB allows indexing for performance optimization. If a unique index exists on a field, duplicate inserts will fail.

To create a unique index:

```
db.employees.createIndex({ id: 1 }, { unique: true })
```

Schema Initialization,

MongoDB is a NoSQL database, meaning it does not enforce a fixed schema like relational databases (SQL-based). However, initializing a schema in MongoDB typically refers to setting up indexes and inserting initial data into collections.

Why Schema Initialization?

Since MongoDB does not require predefined schemas, schema initialization in this context means:

1. **Clearing old data** – Removing all existing documents from a collection.
2. **Inserting default data** – Adding sample data for testing or initial use.
3. **Creating indexes** – Optimizing searches by indexing key fields.

Steps for Schema Initialization

1. Remove Existing Data

Before inserting fresh data, we clear the issues collection to remove any existing documents:

```
db.issues.remove({});
```

This ensures that the database starts with a clean state.

2. Insert Initial Data

A predefined array of issue objects is inserted using insertMany(), providing a structured starting point.

```
const issuesDB = [
  { id: 1, status: 'New', owner: 'Ravan', effort: 5, created: new
    Date('2019-01-15'), due: undefined, title: 'Error in console when
    clicking Add',
  },
  { id: 2, status: 'Assigned', owner: 'Eddie', effort: 14, created: new Date('2019-01-16'),
  due: new Date('2019-02-01'), title: 'Missing bottom border on panel', },
];
db.issues.insertMany(issuesDB);
```

This helps developers by providing sample data for testing the application.

3. Create Indexes

Indexes are essential for improving query performance. The script creates indexes on:

- id (unique index) to prevent duplicate issue entries.
- status, owner, and created fields for faster filtering and sorting.

db.issues.createIndex({ id: 1 }, { unique: true }); db.issues.createIndex({ status: 1 }); db.issues.createIndex({ owner: 1 }); db.issues.createIndex({ created: 1 });

How to Run the Initialization Script

The script can be executed in different environments:

1. **Local MongoDB instance:** mongo issuetracker scripts/init.mongo.js
2. **MongoDB Atlas (Cloud Database):**
   mongo mongodb+srv://user:pwd@xxx.mongodb.net/issuetracker scripts/init.mongo.js
3. **mLab (Hosted MongoDB Service):**
   mongo mongodb://user:pwd@xxx.mlab.com:33533/issuetracker scripts/init.mongo.js

Reading from MangoDB

In a Node.js application, reading data from MongoDB involves establishing a persistent connection to the database and querying collections for relevant data. This process ensures that the application retrieves real-time data from the database instead of relying on an in-memory array.

Steps for Reading from MangoDB

1. Establishing a Persistent Database Connection

Instead of opening and closing the database connection for each query, we maintain a global connection variable (db). This improves efficiency and ensures smooth handling of multiple API requests.

Define the MongoDB Connection URL

```
const { MongoClient } = require('mongodb');

const url = 'mongodb://localhost/issuetracker'; // Local Database // For Cloud Databases,
use:
// const url = 'mongodb+srv://user:password@xxx.mongodb.net/issuetracker'; Create an Asynchronous
Function to Connect

We use the MongoClient from the mongodb package to connect to the database asynchronously:

let db;

async function connectToDb() { const client = new MongoClient(url, { useNewUrlParser:
  true }); await client.connect(); // Connect to MongoDB console.log('Connected to
  MongoDB at', url);
  db = client.db(); // Store database connection in a global variable }
```

2. Modifying the API to Read from MongoDB

Once the connection is established, we update the API function issueList() to retrieve data from MongoDB instead of an in-memory array.

Updating the issueList() Function async function issueList() { const issues = await db.collection('issues').find({}).toArray(); return issues;
}

- .collection('issues') – Accesses the issues collection in MongoDB.
- .find({}) – Retrieves all documents.
- .toArray() – Converts the result to an array.

3. Starting the Server After Database Connection

Since connectToDb() is an asynchronous function, we need to ensure that the database connection is established before the server starts.

Using an Immediately Invoked Async Function

```
(async function () { try {
        await connectToDb(); // Connect to MongoDB first
        app.listen(3000, function () { console.log('App started on
        port 3000'); });
    } catch (err) { console.log('ERROR:', err);
    }
})();
```

This ensures:

- The database is connected before the server starts.
- Any connection errors are caught and logged.

4. Updating the GraphQL Schema

Since MongoDB automatically adds an _id field to each document, we update the GraphQL schema to include this field:

```
type Issue { _id: ID! id:
    Int!
    ...
}
```

Thi allows clients t access _ whe querying t AP

5. Verifying Data is Read from MongoDB

- Run the server.
- Open the UI and check if the issues are displayed.
- Modify data in MongoDB using the shell:

  db.issues.updateMany({}, { $set: { effort: 100 } });

  Refresh the UI – the updated values should reflect.

Writing to MangoDB

To fully integrate MongoDB into the application, we need to update the Create API so that it writes new issues directly to the database instead of storing them in an in-memory array.

Steps for Writing to MangoDB

1. Generating Unique IDs for Issues

Unlike relational databases, MongoDB does not provide built-in auto-increment sequences. To ensure unique, sequential IDs for issues, we implement a **c**ounter **c**ollection.

Initialize the Counter in MongoDB

Before using the counter, we initialize it in init.mongo.js:

```
print('Inserted', count, 'issues');
db.counters.remove({ _id: 'issues' }); // Remove any existing counter db.counters.insert({ _id: 'issues', current: count }); // Initialize counter
```

- The counter document stores the last assigned issue ID.
- When a new issue is added, the counter is incremented, ensuring unique IDs.

Run the Initialization Script After modifying init.mongo.js, execute: mongo issuetracker scripts/init.mongo.js

2. Implementing a Function to Generate Unique IDs

To increment and fetch the latest issue ID, we define getNextSequence() using MongoDB's findOneAndUpdate**()** method.

Function to Get the Next Sequence Number

```
async function getNextSequence(name) {
const result = await db.collection('counters').findOneAndUpdate(
        { _id: name }, // Find the counter by its name
        { $inc: { current: 1 } }, // Increment the counter by 1
        { returnOriginal: false } // Return the updated document
    );
    return result.value.current; // Return the new issue ID
}
```

- **findOneAndUpdate()** ensures atomic updates.
- **$inc: { current: 1 }** increases the counter.
- **returnOriginal: false** makes sure we get the updated value.

3. Updating the Create API to Write to MongoDB Instead of pushing new issues into an in-memory array, we:

1. **Get a new unique ID using getNextSequence()**.
2. **Insert the new issue into the MongoDB collection**.
3. **Retrieve and return the newly created issue**.

Updating issueAdd() Function async function issueAdd(_, { issue }) {
issue.created = new Date(); // Set creation timestamp
issue.id = await getNextSequence('issues'); // Generate unique ID

const result = await db.collection('issues').insertOne(issue); // Insert into MongoDB
const savedIssue = await db.collection('issues')
.findOne({ _id: result.insertedId }); // Retrieve the saved issue

return savedIssue; // Return the newly added issue
}

How This Works

1. **Generate a new ID** → getNextSequence('issues')
2. **Insert issue into MongoDB → insertOne(issue)**
3. **Fetch the inserted document** → findOne({ _id: result.insertedId })
4. **Return the inserted issue** → Ensures the API response includes database- generated fields.

4. Removing the In-Memory Array

Since we now store issues in MongoDB, we can remove the in-memory array:

const issuesDB = [// This is no longer needed
{ id: 1, status: 'New', owner: 'Ravan', effort: 5, ... },
{ id: 2, status: 'Assigned', owner: 'Eddie', effort: 14, ... },
];

5. Testing the Changes Check if the API Works

1. **Run the server** node server.js

2 **Add a new issue from th U**
3 **Check i th issue persists after a server restart**

Cross-Check Using MongoDB Shell

To verify that the issue is saved in MongoDB:

mongo
use issuetracker db.issues.find().pretty()
This should display all issues, including the newly created one.

Modularization and Webpack

We started to get organized in the previous chapter by changing the architecture and adding checks for coding standards and best practices. In this chapter, we'll take this a little further by splitting the code into multiple files and adding tools to ease the process of development. We'll use Webpack to help us split frontend code into component-based files, inject code into the browser incrementally, and refresh the browser automatically on front-end code changes. That's a perfectly valid thought if you are not too concerned about all these, and instead rely on someone else to give you a template of sorts that predefines the directory structure as well as has configuration for the build tools such as Webpack. This can let you focus on the MERN stack alone, without having to deal with all the tooling. In that case, you have the following options:

- Download the code from the book's GitHub repository (https://github.com/ vasansr/pro-mern-stack-2) as of the end of this chapter and use that as your starting point for your project.
- Use the starter-kit create-react-app (https://github.com/facebook/createreact-app) to start your new React app and add code for your application. But note that create-react-app deals only with the React part of the MERN stack; you will have to deal with the APIs and MongoDB yourself.
- Use mern.io (http://mern.io) to create the entire application's directory structure, which includes the entire MERN stack.s

Back-End Modules

Backend modules help organize and structure code for maintainability, scalability, and reusability. In a Node.js application, modularization is achieved using the require() function for importing and module. exports for exporting functionalities. Below is an explanation of key modules used in the Issue Tracker API, along with how they interact.

1. graphql_date.js (Custom GraphQL Scalar Type)

This module defines a custom GraphQL scalar type for handling date values. It:

- Imports required GraphQL modules.
- Defines a GraphQLScalarType for Date.
- Exports the date scalar for use in the GraphQL schema.

Code:

```
const { GraphQLScalarType } = require('graphql'); const { Kind } = require('graphql/language');

const GraphQLDate = new GraphQLScalarType({ // Implementation details...
});
module.exports = GraphQLDate;
```

2. about.js (Handling API Information)

This module manages the "about" message, which describes the API version. It:

- Stores the aboutMessage.
- Defines tw functions: getMessage() (retrieves t message) an setMessage() (updates it).
- Exports these functions.

Code:

```
let aboutMessage = 'Issue Tracker API v1.0';
function setMessage(_, { message }) {
  aboutMessage = message;
}
function getMessage() {
  return aboutMessage;
}
module.exports = { getMessage, setMessage };
```

3. db.js (Database Connection and ID Generation)

This module handles MongoDB connectivity and ID sequence management. It:

- Connects to MongoDB using connectToDb().
- Provides a function getNextSequence() to generate unique issue IDs.
- Exports these functions along with getDb() to get the database connection.

Code:

```
require('dotenv').config();
const { MongoClient } = require('mongodb');
let db;
async function connectToDb() {
  db = client.db();
}
const url = process.env.DB_URL || 'mongodb://localhost/issuetracker';
const client = new MongoClient(url, { useNewUrlParser: true });
await client.connect();
async function getNextSequence(name) {
  const result = await db.collection('counters').findOneAndUpdate(
    { _id: name },
    { $inc: { current: 1 } },
    { returnOriginal: false }
  );
  return result.value.current;
}
function getDb() {
  return db;
}
module.exports = { connectToDb, getNextSequence, getDb };
```

4. issue.js (Issue Management)

This module handles CRUD operations related to issues. It:

- Uses getDb() to get the database connection.
- Provides list() to retrieve all issues.
- Provides add() to create a new issue.

Code:

```
const { UserInputError } = require('apollo-server-express');
const { getDb, getNextSequence } = require('./db.js');
async function getIssues() {
  const db = getDb();
  return await db.collection('issues').find({}).toArray();
}
async function addIssue(_, { issue }) {
  const db = getDb();
  issue.id = await getNextSequence('issues');
  issue.created = new Date();

  const result = await db.collection('issues').insertOne(issue);
  return db.collection('issues').findOne({ _id: result.insertedId });
}
```

5. api_handler.js (GraphQL Schema and Apollo Server)
This module sets up the GraphQL schema and resolver functions. It:

- Imports necessary resolvers from graphql_date.js, about.js, and issue.js.
- Creates an Apollo Server instance with the schema.
- Exports the function installHandler() to integrate GraphQL with Express.

Code:

```
const { ApolloServer } = require('apollo-server-express'); const GraphQLDate =
require('./graphql_date.js');
const about = require('./about.js'); const issue =
require('./issue.js');

const resolvers = {
  Query: { about: about.getMessage,
    issueList: issue.list,
  },
  Mutation: { setAboutMessage: about.setMessage,
    issueAdd: issue.add,
  },
  GraphQLDate,
};
function installHandler(app) {
  server.applyMiddleware({ app, path: '/graphql' });
cons server = ne ApolloServer({ typeDefs, resolvers } ;
} module.exports = { installHandler };
```

6. **server.js** (Starting the Express Server) This is the entry point of the application. It:

- Imports required modules.
- Connects to the database.
- Sets up the Express server.
- Calls installHandler() to integrate GraphQL.

Code:

```
require('dotenv').config(); const express = require('express'); const {
connectToDb } = require('./db.js'); const { installHandler } =
require('./api_handler.js'); const app = express(); installHandler(app);
const port = process.env.API_SERVER_PORT || 3000;

(async function () { try {
    await connectToDb();
    app.listen(port, () => console.log(`API server started on port ${port}`));
  } catch (err) { console.error('ERROR:', err);
  }
})();
```

Front-End Modules and Webpack

In modern web applications, managing large JavaScript files becomes challenging as the project grows. Traditionally, developers used multiple JavaScript files, including them in an HTML file using <script> tags. However, this approach required careful manual dependency management, making it error-prone and difficult to scale.

To solve this, tools like Webpack and Browserify automate dependency resolution, allowing developers to write modular code while bundling everything into optimized JavaScript files.

Why Use Modules in Front-End Development?

1. **Maintainability:**
   - Breaking the code into smaller, reusable modules makes it easier to understand and manage.
2. **Dependency Management:**
   - Webpack automatically determines module dependencies and includes them in the final bundle.
3. **Performance Optimization:**
   - Webpack minifies and optimizes JavaScript, reducing load times.
4. **Scalability:**
   - Modular code allows teams to collaborate efficiently and extend the application easily.

Setting Up Webpack

Since Webpack is only needed for development, we install it as a **dev dependency**:

cd ui npm install --save-dev webpack@4 webpack-cli@3

To verify the installation:

npx webpack --version

Bundling JavaScript Using Webpack

Let's bundle the App.js file into app.bundle.js using Webpack:

npx webpack public/App.js --output public/app.bundle.js

However, Webpack gives a warning about the missing **mode** option. To remove the warning, specify **development mode**:

npx webpack public/App.js --output public/app.bundle.js --mode development

In **development mode**, Webpack keeps the code readable for debugging. In **production mode**, Webpack minifies and optimizes the code.

Splitting Code into Modules

Instead of writing everything in one file, we **split** the code into separate modules.

For example, we extract the graphQLFetch function into a new file.

Step 1: Create graphQLFetch.js graphQLFetch.js

```
const dateRegex = new RegExp('^\\d\\d\\d\\d-\\d\\d-\\d\\d');

function jsonDateReviver(key, value) {
  if (dateRegex.test(value)) return new Date(value); return value;
}

export default async function graphQLFetch(query, variables = {}) { // Fetch implementation
  here
}
```

Step 2: Import the Module in App.jsx

Instead o defining t function inside App.jsx, w import i

import graphQLFetch from './graphQLFetch.js';

ES6 Modules in Webpack Importing Modules

- **Using require() (CommonJS - Older Method)**: const graphQLFetch = require('./graphQLFetch.js');
- **Using import (ES6 - Modern Method)**:

  import graphQLFetch from './graphQLFetch.js';

  - Webpack resolves dependencies automatically.

Exporting Modules

- **Named Export** (for multiple functions): export function graphQLFetch() { ... }
  - Imported as:

    import { graphQLFetch } from './graphQLFetch.js';
- **Default Export** (for a single function):

  export default function graphQLFetch() { ... } ○ Imported as:

    import graphQLFetch from './graphQLFetch.js';

Updating index.html to Use the Bundle

Since Webpack generates app.bundle.js, update index.html to use it instead of App.js:

```
<script src="/env.js"></script>
<script src="/app.bundle.js"></script>
```

This ensures the browser loads the bundled file containing all required modules.

Final Webpack Build Process Steps:

1. **Compile JS usin Babel** (i required):
   npm run compile
2. **Run Webpack to bundle files**:
   npx webpack public/App.js --output public/app.bundle.js --mode development
3. **Test the application** in the browser.

Benefits of Using Webpack

Automatic Dependency Management – No need to manually track script order.
Improved Performance – Bundling reduces HTTP requests and optimizes code.
Code Splitting – Supports loading parts of the application asynchronously.
Faster Development – Webpack watches for changes and updates the bundle automatically.
Handles Static Assets – Can bundle CSS, images, and fonts.

1. **Transform & Bundle with Webpack**
   - Webpack combines transformation and bundling using **loaders** (e.g., babel-loader for Babel).
   - Installed Babel loader: npm install --save-dev babel-loader@8

2. **Webpack Configuration (webpack.config.js)**

○ **Entry**: Specifies the main file (App.jsx).
○ **Output**: Generates app.bundle.js in the public directory.
○ **Module Rules**: Uses Babel loader to transform .jsx and .js files. ○ Example:

```
const path = require('path'); module.exports = {
  mode:       'development',       entry:
  './src/App.jsx', output: {
    filename:  'app.bundle.js',  path:  path.resolve(  dirname,
    'public'),
  },                          —
  module: { rules: [
      { test: /\.jsx?$/, use: 'babel-loader' },
    ],
  },
};
```

3. **Running Webpack** ○ Compile once: npx webpack
   - Watch for changes:
     npx webpack --watch
4. **Updating npm Scripts (package.json)** ○ **Production Build**: "compile": "webpack --mode production" ○ **Watch Mode**: "watch": "webpack --watch"
5. **Modularizing Components** ○ **Separated Components**:
   - IssueList.jsx, IssueFilter.jsx, IssueTable.jsx, IssueAdd.jsx, graphQLFetch.js ○ **App.jsx** now only imports IssueList.jsx and renders it.
6. **Final Testing** ○ Running npm run watch ensures all files are transformed and bundled.
   - The application should work as before but with improved modularity.

Bundling Libraries with Webpack

Previously, we included third-party libraries (like React, ReactDOM, and Babel polyfills) using CDNs in **index.html**. However, this approach has drawbacks:

- **Dependency on CDN availability** – If the CDN is down, the app won't load.
- **Performance issues** – Every time the app loads, the browser fetches these libraries.
- **No caching benefit** – Even if only the app code changes, all scripts are refetched.

Installing Client-Side Libraries with npm

To manage these libraries within our project, we install them using npm:

```
$ cd ui
$ npm install react@16 react-dom@16
$ npm install prop-types@15
$ npm install whatwg-fetch@3
$ npm install babel-polyfill@6
```

After installation, we **import** these libraries in our React files instead of relying on CDNs.

Optimizing Bundling in Webpack

1. Single Bundle Issue

Initially, Webpack bundled everything (including libraries) into app.bundle.js, making the file large (>1MB). This meant that even minor application changes required users to re-download the entire file.

2. Creating Separate Bundles

To improve efficiency, we split the libraries into a separate **vendor bundle** using Webpack's splitChunks optimization. This results in: ○ vendor.bundle.js (for third-party libraries) ○ app.bundle.js (for our application code)

3. Webpack Configuration Changes

- ○ **Exclude node_modules** from transformations since they are already optimized.
- ○ **Modify entry and output** to support multiple bundles.
- ○ **Enable splitChunks** to separate dependencies.

```
module.exports = {
  mode: 'development', entry: { app:
  './src/App.jsx' }, output: {
    filename: '[name].bundle.js', // Generates app.bundle.js and
vendor.bundle.js path: path.resolve(_dirname, 'public'),
  },
  module: { rules: [
      { test: /\.jsx?$/,
        exclude: /node_modules/, use: 'babel-loader',
      },
    ],
  },
  optimization: { splitChunks: {
    name: 'vendor', chunks: 'all',
    },
  },
};
```

4. **Updating index.html**
   - **Remove CDN links**.
   - **Include vendor.bundle.js before app.bundle.js**.

```
<script src="/vendor.bundle.js"></script>
<script src="/app.bundle.js"></script>
```

Benefits of Splitting Bundles

- **Performance Boost**: The browser caches vendor.bundle.js, so changes to application code only require reloading app.bundle.js.
- **Reduced Load Times**: Instead of re-downloading everything, only the necessary updates are fetched.
- **Offline Resilience**: The app is not dependent on CDNs for libraries.

Hot Module Replacement

Problem with Webpack Watch Mode

- Webpack's watch mode detects file changes and recompiles, but **you must manually refresh** the browser to see updates.
- Refreshing **too soon** might load an outdated bundle.
- Requires **an extra terminal** for running npm run watch.

Solution: Hot Module Replacement (HMR)

HMR updates **only the changed modules** in the browser **without refreshing the entire page**, preserving application state.

Key Benefits:

- **No need to refresh manually**
- **Retains state** (e.g., text in input fields stays)
- **Saves time** by only updating changed code

Implementing HMR in an Express-based UI Server

1. **Install Middleware Packages**

```
npm install --save-dev webpack-dev-middleware@3 webpack-hot- middleware@2
```

2. **Modify webpack.config.js** Change entry point to an **array** to add HMR support:

```
entry: { app: ['./src/App.jsx'] }
```

Add HMR entry point: config.entry.app.push('webpack-hot-middleware/client'); Enable HMR plugin: config.plugins.push(new webpack.HotModuleReplacementPlugin());

3. **Modify uiserver.js to Include HMR Middleware**

```
if (enableHMR && process.env.NODE_ENV !== 'production') { const webpack =
require('webpack'); const devMiddleware = require('webpack-dev-middleware');
const hotMiddleware = require('webpack-hot-middleware'); const config =
require('./webpack.config.js');

config.entry.app.push('webpack-hot-middleware/client');
config.plugins = config.plugins || [];
config.plugins.push(new webpack.HotModuleReplacementPlugin());

const compiler = webpack(config); app.use(devMiddleware(compiler));
app.use(hotMiddleware(compiler)); }
```

4. **Modify App.jsx to Accept HMR**

```
if (module.hot) { module.hot.accept();

}
```

Running the Server

Command	Mode	Behavior
`npm run compile + npm run start`	Production	Uses pre-built bundles from `public/`.
`npm run start`	Development	HMR enabled, updates without refresh.
`npm run watch + npm run start`	Dev/Prod (HMR disabled)	Watches files but requires refresh.

Handling Component Reload Issues

- By default, HMR reloads the **entire React component tree**, **losing local state**.
- **Solution:** Use **react-hot-loader** (not implemented in this case).

Debugging DefinePlugin:

- The unpleasant thing about compiling files is that the original source code gets lost, and if you have to put breakpoints in the debugger, it's close to impossible, because the new code is hardly like the original.
- Creating a bundle of all the source files makes it worse, because you don't even know where to start. Fortunately, Webpack solves this problem by its ability to give you source maps, things that contain your original source code as you typed it in.
- The source maps also connect the line numbers in the transformed code to your original code. Browsers' development tools understand source maps and correlate the two, so that breakpoints in the original source code are converted breakpoints in the transformed code.
- Webpack configuration can specify what kind of source maps can be created along with the compiled bundles. A single configuration parameter called devtool does the job.
- The kind of source maps that can be produced varies, but the most accurate

(and the slowest) is generated by the value source-map. For this application, because the UI code is small enough, this is not discernably slow, so let's use it as the value for devtool. The changes to webpack.config.js in the UI directory are shown in Listing 8-29

Listing 8-29. ui/webpack.config.js: Enable Source Map

```
...
  optimization: {
    ...
  },
  devtool: 'source-map'
};
...
```

If you are using the HMR-enabled UI server, you should see the following output in the console that is running the UI server:

```
webpack built dc6a1e03ee249e546ffb in 2964ms
[wdm]: Hash: dc6a1e03ee249e546ffb
Version: webpack 4.23.1
Time: 2964ms
Built at: 10/27/2018 12:08:12 AM
                Asset      Size  Chunks             Chunk Names
       app.bundle.js  54.2 KiB     app  [emitted]  app
   app.bundle.js.map  41.9 KiB     app  [emitted]  app
    vendor.bundle.js  1.26 MiB  vendor  [emitted]  vendor
vendor.bundle.js.map   1.3 MiB  vendor  [emitted]  vendor
Entrypoint app = vendor.bundle.js vendor.bundle.js.map app.bundle.js app.bundle.js.map
[0] multi ./src/App.jsx webpack-hot-middleware/client 40 bytes {app} [built]
[./node_modules/ansi-html/index.js] 4.16 KiB {vendor} [built]
[./node_modules/babel-polyfill/lib/index.js] 833 bytes {vendor} [built]
...
```

As you can see, apart from the package bundles, there are accompanying maps with the extension .map. Now, when you look at the browser's Developer Console, you will be able to see the original source code and be able to place breakpoints in it. A sample of this in the Chrome browser is shown in Figure 8-1

Figure 8-1. *Breakpoints in the original source code using source maps*

If you are using the Chrome or Firefox browser, you will also see a message in the console asking you to install the React Development Tools add-on. You can find installation instructions for these browsers at: https://reactjs.org/blog/2015/09/02/new-react-developer-tools.html

This add-on provides the ability to see the React components in a hierarchical manner like the DOM inspector. For example, in a Chrome browser, you'll find a React tab in the developer tools. Figure 8-2 shows a screenshot of this add-on.

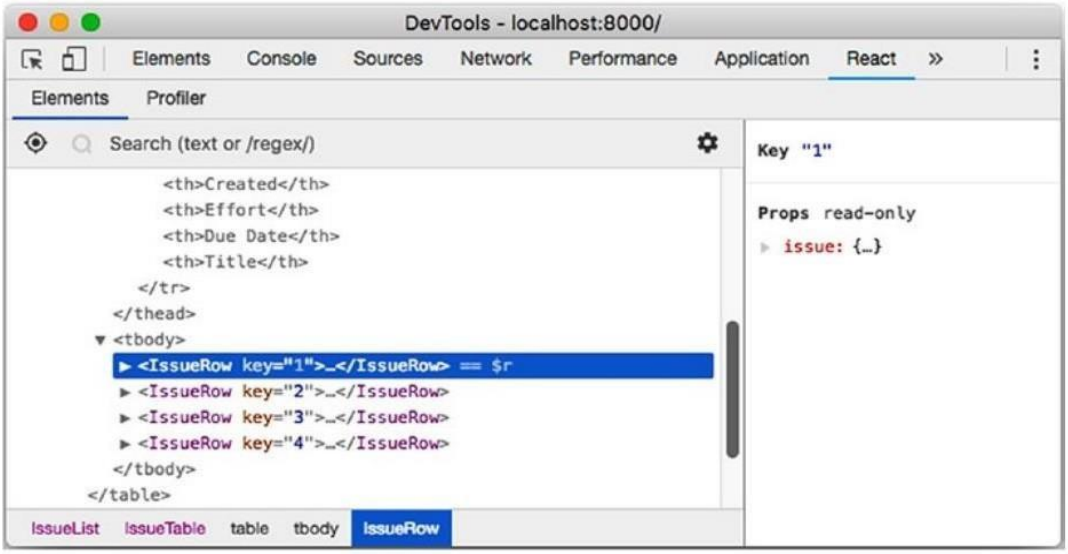

Figure 8-2. *React Developer Tools in the Chrome browser*

Define Plugin: Build Configuration

You may not be comfortable with the mechanism that we used for injecting the environment variables in the front-end: a generated script like env.js. For one, this is less efficient than generating a bundle that already has this variable replaced wherever it needs to be replaced. The other is that a global variable is normally frowned upon, since it can clash with global variables from other scripts or packages. Fortunately,

```
...
  plugins: [
    new webpack.DefinePlugin({
      __UI_API_ENDPOINT__: "'http://localhost:3000/graphql'",
    })
  ],
...
```

Now, within the code for App.jsx, instead of hard-coding this value, the __UI_API_ENDPOINT__ string can be used like this (note the absence of quotes; it is provided by the variable itself):

```
...
    const response = await fetch(__UI_API_ENDPOINT__, {
...
```

When Webpack transforms and creates a bundle, the variable will be replaced in the source code, resulting in the following:

```
...
    const response = await fetch('http://localhost:3000/graphql', {
...
```

Within webpack.config.js, you can determine the value of the variable by using dotenv and an environment variable instead of hard-coding it there:

```
...
require('dotenv').config();
...
    new webpack.DefinePlugin({
      __UI_API_ENDPOINT__: `'${process.env.UI_API_ENDPOINT}'`,
    })
...
```

there is an option. We will not be using this mechanism for injecting environment variables, but I have discussed it here so that it gives you an option to try out and adopt if convenient.

To replace variables at build time, Webpack's DefinePlugin plugin comes in handy. As part of webpack.config.js, the following can be added to define a predefined string with the value like this:

Production Optimization

1 **Bundle Size & Performance**

- Webpack minifies JavaScript in production mode.
- Initial vendor bundle size is small (~200KB) but grows as features are added.
- Large bundle sizes can trigger Webpack warnings about performance impact.

2 Handling Large Bundles

- **Frequent-use apps (e.g., Issue Tracker):** Browser caching reduces concerns.
- **Infrequent-use apps:** Need better optimization to improve page load times.
- **Solution:** Use **code splitting** and **lazy loading** to load scripts only when required.

3 Lazy Loading Strategy

- Load only essential scripts upfront.
- Postpone loading non-critical components until needed. Useful in **server-rendered** React apps.

4 Browser Caching Issues

- Older browsers (e.g., Internet Explorer) may aggressively cache outdated scripts.
- Modern browsers usually check for updates, but explicit cache-busting may be needed.

5 Cache Busting Solution

- Use **content hashes** in script file names to force browsers to load new versions.
- Webpack can generate unique file names based on content changes.

6 Managing Auto-Generated Script Names

- Need to update index.html with new script names dynamically.
- **Solution:** Use **HTMLWebpackPlugin** to generate index.html automatically.

7 Further Reading

- Webpack Code Splitting: webpack.js.org/guides/code-splitting
- Webpack Lazy Loading: webpack.js.org/guides/lazy-loading
- Webpack Caching: webpack.js.org/guides/caching
- HTMLWebpackPlugin: webpack.js.org/plugins/html-webpack-plugin