

## Agenda for Today's Class

1. How to Run JS program using Node JS

### 2. Inheritance (30 minutes)

#### - Define inheritance and its purpose in OOP

- Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit properties and methods from another class. It establishes a hierarchical relationship between classes, enabling code reuse and promoting modularity in software design.
- The purpose of inheritance in OOP is to facilitate the creation of new classes (called subclasses or derived classes) that inherit characteristics (such as attributes and behaviors) from existing classes (called super classes or base classes). The derived classes can then extend or modify the inherited properties and methods, as well as add new ones specific to their own requirements.

#### Key purposes of inheritance in OOP include:

- **Code Reuse:** Inheritance allows subclasses to inherit attributes and methods from super classes, reducing code duplication. This promotes efficiency and maintainability by leveraging existing code and building upon it, rather than starting from scratch.
- **Hierarchy and Organization:** Inheritance establishes a hierarchical relationship between classes, enabling a structured and organized class hierarchy. Classes can be grouped based on their common characteristics and behaviors, forming a logical and intuitive structure for managing complex systems.
- **Modularity and Extensibility:** Inheritance promotes modularity by separating different aspects of a system into distinct classes. Subclasses can be created to extend or modify the behavior of the inherited super class, allowing for incremental development and making it easier to add new features or adapt existing ones.
- **Polymorphism:** Inheritance is closely tied to polymorphism, another important OOP concept. Polymorphism allows objects of different classes to be treated as objects of a common super class, providing flexibility and allowing for more generic and reusable code. By leveraging inheritance, developers can create a more organized, reusable, and extensible codebase. It enables the development of complex systems by building upon existing classes, encapsulating common behaviors and characteristics, and providing a structured framework for software development.

#### - Discuss the concept of super class and subclass.

Superclass:

- A superclass, also known as a base class or parent class, is a class that is extended by other classes.

- It represents a more general and abstract concept or category.
- Superclasses encapsulate common attributes and behaviors that are shared among its subclasses.
- Superclasses define a blueprint or template that subclasses can inherit from and extend.

```
class Animal {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  eat() {
    console.log(this.name + " is eating.");
  }
}
```

- In this example, the `Animal` class is a superclass that represents a generic animal. It has properties like `name` and `age` and a method `eat()`.

Subclass:

- A subclass is a class that inherits properties and methods from a superclass.
- It represents a more specific and specialized concept or category.
- Subclasses extend the functionality of the superclass by adding or modifying properties and methods.
- Subclasses can have their own unique attributes and behaviors in addition to the inherited ones.

Example:

```
class Dog extends Animal {
  constructor(name, age, breed) {
    super(name, age);
    this.breed = breed;
  }

  bark() {
    console.log(this.name + " is barking.");
  }
}
```

- In this example, the `Dog` class is a subclass that extends the `Animal` superclass. It adds a `breed` property and a `bark()` method to the inherited properties and methods from `Animal`. The `super()` keyword is used in the constructor to call the superclass constructor and initialize the inherited properties.
- The superclass-subclass relationship allows for hierarchical organization and code reuse. The superclass provides a common structure and behavior shared by multiple subclasses. Subclasses inherit the characteristics of the superclass and can add their own specific features. This concept promotes modularity, flexibility, and extensibility in object-oriented programming.

## - Demonstrate examples of inheritance in JavaScript code.

### 1. Single Inheritance:

```
// Parent class
class Animal {
  constructor(name) {
    this.name = name;
  }

  eat() {
    console.log(this.name + " is eating.");
  }
}

// Child class
class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
}

// Creating instances
const dog = new Dog("Buddy", "Labrador");
dog.eat(); // Output: Buddy is eating.
```

### 2. Multi Level Inheritance

```
// Parent class
class Animal {
  constructor(name) {
    this.name = name;
  }

  eat() {
    console.log(this.name + " is eating.");
  }
}

// Child class
class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
}

// Grandchild class
class Puppy extends Dog {
  constructor(name, breed, age) {
    super(name, breed);
  }
}
```

```

        this.age = age;
    }
}

// Creating instances
const puppy = new Puppy("Charlie", "Golden Retriever", 1);
puppy.eat(); // Output: Charlie is eating.

```

### 3. Hierarchical Inheritance

Hierarchical inheritance in JavaScript refers to a scenario where multiple subclasses inherit from a common superclass. Each subclass inherits properties and methods from the common superclass while having the flexibility to add its own unique properties and methods. Here's an example of hierarchical inheritance in JavaScript:

```

// Parent class
class Animal {
    constructor(name) {
        this.name = name;
    }

    eat() {
        console.log(this.name + " is eating.");
    }
}

// Child class 1
class Dog extends Animal {
    bark() {
        console.log(this.name + " is barking.");
    }
}

// Child class 2
class Cat extends Animal {
    meow() {
        console.log(this.name + " is meowing.");
    }
}

// Creating instances
const dog = new Dog("Buddy");
dog.eat(); // Output: Buddy is eating.
dog.bark(); // Output: Buddy is barking.

const cat = new Cat("Whiskers");
cat.eat(); // Output: Whiskers is eating.
cat.meow(); // Output: Whiskers is meowing.

```

### 3. Polymorphism (30 minutes)

#### - Define polymorphism and its significance in OOP.

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass or interface. It refers to the ability of objects to take on multiple forms or exhibit different behaviors depending on the context in which they are used. In OOP, polymorphism is significant because it promotes code flexibility, reusability, and extensibility. Here are some key aspects of polymorphism and its significance:

- i. Code Flexibility
- ii. Method Overriding
- iii. Dynamic Binding

#### Code Flexibility

```
class Shape {
  calculateArea() {
    // Common area calculation logic
  }
}

class Circle extends Shape {
  calculateArea() {
    // Circle-specific area calculation logic
  }
}

class Rectangle extends Shape {
  calculateArea() {
    // Rectangle-specific area calculation logic
  }
}

// Usage
function printArea(shape) {
  console.log(shape.calculateArea());
}

const circle = new Circle();
const rectangle = new Rectangle();

printArea(circle);    // Output: Circle-specific area calculation result
printArea(rectangle); // Output: Rectangle-specific area calculation result
```

In this example, the Shape class represents a common superclass with a calculateArea() method. The Circle and Rectangle classes are subclasses that extend the Shape class and provide their own

implementation of the `calculateArea()` method. The `printArea()` function demonstrates code flexibility by accepting objects of different classes as arguments and calling the `calculateArea()` method on them. It treats objects of different classes as objects of the common type `Shape`, allowing for a generalized approach to calculate and print the area.

## Method Overriding

```
class Vehicle {
  startEngine() {
    console.log("Engine started.");
  }
}

class Car extends Vehicle {
  startEngine() {
    console.log("Car engine started.");
  }
}

class Motorcycle extends Vehicle {
  startEngine() {
    console.log("Motorcycle engine started.");
  }
}

// Usage
const car = new Car();
const motorcycle = new Motorcycle();

car.startEngine();      // Output: Car engine started.
motorcycle.startEngine(); // Output: Motorcycle engine started.
```

In this example, the `Vehicle` class has a `startEngine()` method. The `Car` and `Motorcycle` classes are subclasses that inherit from the `Vehicle` class and provide their own implementation of the `startEngine()` method. When calling the `startEngine()` method on instances of `Car` and `Motorcycle`, the overridden method in the respective subclasses is invoked, providing the specific behavior for starting the engine.

## Dynamic Binding

```
class Shape {
  draw() {
    console.log("Drawing a shape.");
  }
}

class Circle extends Shape {
  draw() {
    console.log("Drawing a circle.");
  }
}
```

```

class Rectangle extends Shape {
  draw() {
    console.log("Drawing a rectangle.");
  }
}

// Usage
function drawShape(shape) {
  shape.draw();
}

const circle = new Circle();
const rectangle = new Rectangle();

drawShape(circle);    // Output: Drawing a circle.
drawShape(rectangle); // Output: Drawing a rectangle.

```

In this example, the Shape class has a draw() method. The Circle and Rectangle classes are subclasses that inherit from Shape and override the draw() method with their specific implementation. The drawShape() function demonstrates dynamic binding by accepting objects of different classes as arguments and calling the draw() method. The appropriate implementation is determined at runtime based on the actual type of the object being referenced, allowing for dynamic behavior based on the specific object.

### Polymorphic Collections and APIs

```

class Animal {
  makeSound() {
    console.log("Animal makes a sound.");
  }
}

class Dog extends Animal {
  makeSound() {
    console.log("Dog barks.");
  }
}

class Cat extends Animal {
  makeSound() {
    console.log("Cat meows.");
  }
}

// Usage
const animals = [new Dog(), new Cat()];

animals.forEach((animal) => {
  animal.makeSound();
});

```

In this example, the Animal class has a makeSound() method. The Dog and Cat classes are subclasses that inherit from Animal and override the makeSound() method. The animals array holds instances of Dog and Cat. The forEach() method is used to iterate over the array, and the makeSound() method is invoked on each object. Despite having different classes, the objects in the array are treated as objects of a common type (Animal) and can be called using a common interface.

```
class Shape {
  calculateArea() {
    // Common area calculation logic
  }
}

class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }

  calculateArea() {
    return Math.PI * this.radius ** 2;
  }
}

class Rectangle extends Shape {
  constructor(width, height) {
    super();
    this.width = width;
    this.height = height;
  }

  calculateArea() {
    return this.width * this.height;
  }
}

// Usage
const circle = new Circle(5);
const rectangle = new Rectangle(4, 6);

console.log(circle.calculateArea()); // Output: 78.53981633974483
console.log(rectangle.calculateArea()); // Output: 24
```

#### 4. Encapsulation (20 minutes)

- Define encapsulation and its role in OOP.



- Encapsulation is a fundamental principle in object-oriented programming (OOP) that combines data and the methods (or functions) that operate on that data into a single unit called a class. It involves bundling related data and behavior together, and controlling access to that data through methods. The main purpose of encapsulation is to hide the internal details of an object and provide a well-defined interface for interacting with the object.

**Encapsulation in OOP serves several important roles:**

- **Data Protection:** Encapsulation provides a way to protect the internal state of an object by hiding its data from direct external access. The internal data is encapsulated within the object and can only be accessed and modified through controlled methods, also known as getters and setters. This ensures that the data remains in a valid and consistent state, preventing unauthorized or unintended modifications.
- **Abstraction:** Encapsulation allows the concept of abstraction by exposing only the necessary information and hiding the implementation details. The class interface defines a set of public methods that provide a high-level abstraction of the object's behavior. The internal workings of the class can be modified without affecting the code that uses the class, as long as the public interface remains unchanged. This promotes modular design and reduces the impact of changes in the internal implementation.
- **Code Organization and Maintenance:** Encapsulation helps in organizing code into logical units, making it easier to manage and maintain. By bundling related data and behavior together, encapsulation improves code readability and reduces complexity. It allows for the creation of reusable and modular components, making code development and maintenance more efficient.
- **Access Control:** Encapsulation provides control over the accessibility of data and methods within a class. By using access modifiers such as private, protected, and public, encapsulation defines the visibility and availability of class members. Private members are accessible only within the class, protected members are accessible within the class and its subclasses, and public members are accessible from outside the class. Access control enhances security, enforces data integrity, and encapsulates the implementation details of a class.
- **Information Hiding:** Encapsulation supports information hiding by encapsulating the internal details of an object. It allows for the separation of interface and implementation, where the

internal workings of an object are hidden from the outside world. This provides a level of abstraction and prevents external entities from relying on internal implementation details, making it easier to modify and evolve the internal structure without affecting other parts of the code.

- In summary, encapsulation is a key principle in OOP that combines data and methods into a single unit, the class. It provides data protection, abstraction, code organization, access control, and information hiding. Encapsulation plays a vital role in creating modular, maintainable, and secure code by hiding internal details and exposing a well-defined interface for interacting with objects.

**- Explain how encapsulation can be achieved in JavaScript using closures and modules.**

Closures: Closures in JavaScript allow for the creation of a private scope, where variables and functions can be defined. By enclosing data and functions within a closure, they become inaccessible from the outside scope, effectively achieving encapsulation.

```
function createCounter() {
  let count = 0;

  return {
    increment: function() {
      count++;
    },
    decrement: function() {
      count--;
    },
    getCount: function() {
      return count;
    }
  };
}

const counter = createCounter();
counter.increment();
counter.increment();
console.log(counter.getCount()); // Output: 2
```

In the above example, the `createCounter()` function returns an object with three methods: `increment()`, `decrement()`, and `getCount()`. The `count` variable is encapsulated within the closure of the `createCounter()` function and is not accessible from the outside scope. This allows for data hiding and controlled access to the `count` variable through the returned methods, ensuring encapsulation.

**Modules:** Modules in JavaScript provide a way to organize code into self-contained units, each with its own private variables and functions. Modules encapsulate related functionality, allowing the selective exposure of public methods while keeping internal implementation details private.

```

const myModule = (function() {
    let privateVariable = 'Hello';

    function privateFunction() {
        console.log('This is a private function');
    }

    function publicFunction() {
        console.log('This is a public function');
    }

    return {
        publicFunction: publicFunction
    };
})();

myModule.publicFunction(); // Output: This is a public function
myModule.privateVariable; // undefined (private variable is not accessible)
myModule.privateFunction(); // Error (private function is not accessible)

```

In the above example, an immediately-invoked function expression (IIFE) is used to create a module. The private variables (**privateVariable**) and functions (**privateFunction**) are encapsulated within the module's scope and are not directly accessible from the outside. Only the **publicFunction** is exposed and can be accessed through the module object returned by the IIFE. This achieves encapsulation by keeping the internal details private while providing controlled access to public methods.

Closures and modules in JavaScript offer powerful mechanisms for achieving encapsulation by creating private scopes and controlling access to variables and functions. Closures provide a way to encapsulate data within functions, while modules provide a higher level of encapsulation by organizing code into self-contained units with private and public members. These concepts promote encapsulation, information hiding, and modular design in JavaScript applications.

**- Discuss the benefits of encapsulation, such as data hiding and code organization.**

Encapsulation offers several benefits in object-oriented programming, including data hiding and code organization. Let's explore these benefits in more detail:

- **Data Hiding and Protection:**
  - Encapsulation allows for the hiding of internal data within objects. By encapsulating data and providing controlled access through methods, encapsulation protects the data from direct external modification. This prevents unauthorized or unintended changes to the object's state, ensuring data integrity. Data hiding promotes information hiding, where the internal details of an object are concealed, and only a well-defined interface

is exposed. This helps in reducing complexity and isolating the impact of changes, as the external code relies on the public interface rather than the internal implementation.

```
class BankAccount {
  constructor(accountNumber) {
    this._accountNumber = accountNumber; // Private data
    this._balance = 0; // Private data
  }

  deposit(amount) {
    this._balance += amount;
  }

  withdraw(amount) {
    if (amount <= this._balance) {
      this._balance -= amount;
    } else {
      console.log("Insufficient funds");
    }
  }
}

const account = new BankAccount("1234567890");
account.deposit(1000);
account.withdraw(500);
console.log(account._accountNumber); // Output: 1234567890 (direct access, not recommended)
console.log(account._balance); // Output: 500 (direct access, not recommended)
```

- 

In this example, the `BankAccount` class encapsulates the private data `_accountNumber` and `_balance`. These variables are not intended to be accessed directly from the outside. Instead, the public methods `deposit()` and `withdraw()` provide controlled access to manipulate the data. Encapsulation protects the data from unauthorized modifications and ensures that the balance cannot be directly modified without going through the appropriate methods.

- Increased Security:
  - Encapsulation contributes to improved security by hiding sensitive data and providing controlled access to it. Private data and methods can only be accessed within the class or object that encapsulates them, preventing unauthorized access from external code. By controlling access through well-defined public interfaces, encapsulation helps enforce data security and maintain confidentiality.

```
class User {
  constructor(name, email, password) {
    this._name = name; // Private data
    this._email = email; // Private data
    this._password = password; // Private data
  }
}
```

```

login(email, password) {
  if (email === this._email && password === this._password) {
    console.log("Login successful");
  } else {
    console.log("Invalid email or password");
  }
}
}

const user = new User("John Doe", "john@example.com", "password123");
user.login("john@example.com", "password123");
console.log(user._password); // Output: password123 (direct access, not recommended)

```

In this example, the `User` class encapsulates private data such as the user's name, email, and password. By keeping these data private, encapsulation provides a level of security. The `login()` method allows users to authenticate by providing their email and password. The password is not directly accessible from the outside, reducing the risk of exposing sensitive information.

- Code Organization and Modularity:
  - Encapsulation promotes code organization and modularity by grouping related data and behavior into cohesive units (classes or objects). By encapsulating data and methods within classes, encapsulation allows for better organization and management of code. It enables the creation of self-contained modules or components that can be developed and maintained independently. This modular design facilitates code reuse, reduces code duplication, and enhances code maintainability.

```

class ShoppingCart {
  constructor() {
    this._items = []; // Private data
  }

  addItem(item) {
    this._items.push(item);
  }

  removeItem(item) {
    const index = this._items.indexOf(item);
    if (index !== -1) {
      this._items.splice(index, 1);
    }
  }

  getItems() {
    return this._items;
  }
}

const cart = new ShoppingCart();
cart.addItem("Item 1");
cart.addItem("Item 2");
console.log(cart.getItems()); // Output: ["Item 1", "Item 2"]
console.log(cart._items); // Output: ["Item 1", "Item 2"] (direct access, not recommended)

```

- 

In this example, the `ShoppingCart` class encapsulates the private data `_items`, representing the items in the shopping cart. The public methods `addItem()`, `removeItem()`, and `getItems()` provide the interface to interact with the cart. By encapsulating the data and behavior related to the shopping cart, encapsulation organizes the code into a self-contained unit, enhancing code modularity and maintainability.

- Enhanced Code Readability:
  - Encapsulation improves code readability by providing a clear and consistent interface for interacting with objects. By encapsulating data and behavior within objects, encapsulation hides the implementation details and exposes a well-defined set of methods. This makes it easier for other developers to understand and use the code, as they can focus on the high-level functionality provided by the object rather than getting lost in the internal implementation.

```
class Person {
  constructor(name, age) {
    this._name = name; // Private data
    this._age = age; // Private data
  }

  introduce() {
    console.log(`Hi, my name is ${this._name} and I am ${this._age} years old.`);
  }
}

const person = new Person("John Doe", 30);
person.introduce(); // Output: Hi, my name is John Doe and I am 30 years old.
console.log(person._name); // Output: John Doe (direct access, not recommended)
console.log(person._age); // Output: 30 (direct access, not recommended)
```

In this example, the `Person` class encapsulates private data such as the person's name and age. The `introduce()` method provides a clear and concise way to present the person's information. By encapsulating the data and exposing a well-defined public method, encapsulation improves code readability and allows other developers to easily understand and use the class.

- Easy Maintenance and Refactoring:
  - Encapsulation simplifies code maintenance and refactoring. By encapsulating data and behavior within objects, changes to the internal implementation can be made without affecting the code that uses the object, as long as the public interface remains unchanged. This reduces the impact of changes and allows for easier maintenance and evolution of the codebase. Encapsulation also helps in isolating the effects of changes, making it easier to debug and troubleshoot issues.

```

class Rectangle {
  constructor(width, height) {
    this._width = width; // Private data
    this._height = height; // Private data
  }

  getWidth() {
    return this._width;
  }

  setWidth(width) {
    if (width > 0) {
      this._width = width;
    }
  }

  getHeight() {
    return this._height;
  }

  setHeight(height) {
    if (height > 0) {
      this._height = height;
    }
  }

  getArea() {
    return this._width * this._height;
  }
}

const rectangle = new Rectangle(5, 10);
console.log(rectangle.getArea()); // Output: 50

rectangle.setWidth(8);
rectangle.setHeight(12);
console.log(rectangle.getArea()); // Output: 96

```

In this example, the `Rectangle` class encapsulates the private data `_width` and `_height`. The class provides public methods to get and set the width and height, ensuring that valid values are assigned.

The `getArea()` method calculates the area of the rectangle. Encapsulation allows for easy maintenance and refactoring of the code. If the internal implementation of the rectangle needs to change, such as using a different formula to calculate the area, only the `getArea()` method needs to be modified, while the external code that uses the class remains unaffected.

- In summary, encapsulation offers benefits such as data hiding, increased security, code organization, modularity, enhanced code readability, and easier maintenance. It promotes code reusability, isolates the impact of changes, and provides a clear interface for interacting with

objects. Encapsulation is a fundamental principle in object-oriented programming that helps create robust, maintainable, and secure software systems.

## 5. Abstraction (20 minutes)

### - Define abstraction and its importance in software development.

- Abstraction is a fundamental concept in software development that involves representing complex systems or ideas in a simplified and generalized manner. It involves hiding unnecessary details and exposing only essential information and functionality to the users. Abstraction provides a high-level view or interface that allows users to interact with a system or object without needing to understand its internal complexities.

**The importance of abstraction in software development can be summarized as follows:**

- **Simplifies Complexity:** Abstraction simplifies complex systems by breaking them down into manageable and understandable components. It allows developers to focus on the essential aspects of a system while hiding unnecessary implementation details. This simplification makes the system more comprehensible, reduces cognitive load, and facilitates the development process

```
class Item {
  constructor(price, quantity) {
    this.price = price;
    this.quantity = quantity;
  }

  getTotalPrice() {
    return this.price * this.quantity;
  }
}

const items = [
  new Item(10, 2),
  new Item(15, 3),
  new Item(5, 4)
];

function calculateTotalPrice(items) {
  let totalPrice = 0;
  items.forEach(item => {
    totalPrice += item.getTotalPrice();
  });
  return totalPrice;
}

console.log(calculateTotalPrice(items)); // Output: 85
```



- The Item class represents an item with a price and quantity. It has a getTotalPrice() method that calculates and returns the total price of the item based on the price and quantity.
- The calculateTotalPrice() function takes an array of Item objects and iterates over each item using forEach(). It calls the getTotalPrice() method on each item and accumulates the total price.
- The function returns the total price of all the items.
- **Manages Complexity:** In large software systems, complexity can quickly become overwhelming. Abstraction helps manage complexity by dividing the system into modules, classes, or functions, each responsible for a specific task or abstraction level. By encapsulating complexity within these abstractions, it becomes easier to reason about, test, and maintain the codebase.

```
class OrderProcessor {
  processOrder(order) {
    // Process order logic
    console.log(`Processing order ${order}`);
  }
}

const orderProcessor = new OrderProcessor();
orderProcessor.processOrder("12345"); // Output: Processing order 12345
```

- The OrderProcessor class encapsulates the logic for processing an order.
- It has a processOrder() method that takes an order parameter and logs a message indicating the processing of the order
- **Enhances Reusability:** Abstraction promotes code reuse by providing well-defined interfaces that can be utilized in various contexts. By abstracting away the implementation details, the interface becomes a contract that can be relied upon by other parts of the system or by external code. This enables the development of modular, reusable components, libraries, and frameworks.

```
class DataFetcher {
  fetchData(url) {
    // Fetch data from the specified URL
    console.log(`Fetching data from ${url}`);
  }
}

const dataFetcher = new DataFetcher();
dataFetcher.fetchData("https://example.com/api/data"); // Output: Fetching data from
https://example.com/api/data
```

In this example, the abstraction of a `DataFetcher` class provides a reusable component for fetching data from a specified URL. By encapsulating the data fetching logic within the class, the abstraction can be easily used in different parts of the system without duplicating the implementation code.

- **Facilitates Modularity:** Abstraction supports the creation of modular software systems. Modules encapsulate related functionality and provide a clean boundary that separates the internal implementation from external interactions. This modularity improves code organization, promotes separation of concerns, and allows for independent development and maintenance of different parts of the system.

```
class InputValidator {
  validateInput(input) {
    // Input validation logic
    console.log(`Validating input ${input}`);
  }
}

const inputValidator = new InputValidator();
inputValidator.validateInput("123"); // Output: Validating input 123
```

- The `InputValidator` class encapsulates the logic for validating input.
- It has a `validateInput()` method that takes an input parameter and logs a message indicating the validation of the input.

- **Supports Flexibility and Adaptability:** Abstraction enables flexibility and adaptability by decoupling different components of a system. By relying on abstractions rather than concrete implementations, it becomes easier to introduce changes or replace components without affecting other parts of the system. This promotes maintainability, extensibility, and the ability to adapt to evolving requirements.

```
class NotificationService {
  sendNotification(notification) {
    // Send notification using a specific service
    console.log(`Sending notification: ${notification}`);
  }
}

const notificationService = new NotificationService();
notificationService.sendNotification("New message"); // Output: Sending notification: New message
```

- The `NotificationService` class provides a generic interface for sending notifications.
- It has a `sendNotification()` method that takes a notification parameter and logs a message indicating the sending of the notification.

- **Improves Collaboration:** Abstraction serves as a common language that facilitates communication and collaboration among developers. It provides a shared understanding of the system's structure, behavior, and interfaces. By abstracting away low-level details, developers can focus on higher-level concepts and collaborate more effectively.

```
class TaxCalculator {
  calculateTax(income) {
    // Tax calculation logic
    console.log(`Calculating tax for income ${income}`);
  }
}

const taxCalculator = new TaxCalculator();
taxCalculator.calculateTax(50000); // Output: Calculating tax for income 50000
```

- The TaxCalculator class represents a component for calculating taxes based on income.
- It has a calculateTax() method that takes an income parameter and logs a message indicating the calculation of tax based on the income.
- **Enhances Testability:** Abstraction improves testability by allowing for isolation and unit testing of individual components. With well-defined abstractions, it becomes easier to write focused tests that verify the behavior of a specific abstraction without considering its internal implementation. This promotes test-driven development, reduces test complexity, and improves the overall quality of the software.
- In summary, abstraction is essential in software development as it simplifies complexity, manages large codebases, enhances reusability, facilitates modularity, supports flexibility and adaptability, improves collaboration, and enhances testability. It enables developers to work with higher-level concepts, promotes code organization, and helps build robust and maintainable software systems.

## 6. Abstract Classes (20 minutes)

- Define abstract classes and their purpose in OOP.

An **abstract class** is a class that cannot be instantiated and serves as a blueprint for other classes. It acts as a base class from which other classes can inherit and extend functionality. Abstract classes

provide a way to define common attributes and methods that are shared among multiple related classes.

**The purpose of abstract classes in object-oriented programming (OOP) is to:**

- Define a Common Interface: Abstract classes allow you to define a common interface or contract that derived classes must adhere to. They specify a set of methods or properties that must be implemented by the subclasses. This ensures consistency and helps enforce a specific structure or behavior across related classes.

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

  // Abstract method
  makeSound() {
    throw new Error("This method must be implemented.");
  }
}

class Dog extends Animal {
  makeSound() {
    console.log("Woof!");
  }
}

class Cat extends Animal {
  makeSound() {
    console.log("Meow!");
  }
}

const dog = new Dog("Max");
dog.makeSound(); // Output: Woof!

const cat = new Cat("Whiskers");
cat.makeSound(); // Output: Meow!
```

•

In this example, the `Animal` class is an abstract class with an abstract method `makeSound()`. The `Dog` and `Cat` classes extend the `Animal` class and provide concrete implementations of the `makeSound()` method. The abstract class defines the common interface by specifying that all subclasses must implement the `makeSound()` method.

- Encapsulate Common Functionality: Abstract classes can encapsulate common functionality or behavior that is shared among multiple subclasses. By defining these common methods or properties in the abstract class, you avoid duplicating code in each subclass. This promotes code reuse, reduces redundancy, and makes maintenance easier.

```
class Shape {
  constructor(color) {
    this.color = color;
  }

  // Abstract method
  draw() {
    throw new Error("This method must be implemented.");
  }
}

class Circle extends Shape {
  draw() {
    console.log(`Drawing a ${this.color} circle.`);
  }
}

class Rectangle extends Shape {
  draw() {
    console.log(`Drawing a ${this.color} rectangle.`);
  }
}

const circle = new Circle("red");
circle.draw(); // Output: Drawing a red circle.

const rectangle = new Rectangle("blue");
rectangle.draw(); // Output: Drawing a blue rectangle.
```

In this example, the `Shape` class is an abstract class with a common property `color` and an abstract method `draw()`. The `Circle` and `Rectangle` classes extend the `Shape` class and provide concrete implementations of the `draw()` method. The abstract class encapsulates the common functionality of having a `color` and defining the `draw()` method.

- Provide Partial Implementation: Abstract classes can provide a partial implementation of certain methods. These methods can contain default behavior or common logic that can be inherited by the subclasses. Subclasses can choose to override these methods to provide their own specific implementation or extend the default behavior provided by the abstract class.

```

class Vehicle {
  constructor() {
    this.isRunning = false;
  }

  start() {
    this.isRunning = true;
  }

  stop() {
    this.isRunning = false;
  }

  // Abstract method
  accelerate() {
    throw new Error("This method must be implemented.");
  }
}

class Car extends Vehicle {
  accelerate() {
    if (this.isRunning) {
      console.log("Car is accelerating.");
    } else {
      console.log("Car cannot accelerate when stopped.");
    }
  }
}

const car = new Car();
car.start();
car.accelerate(); // Output: Car is accelerating.

car.stop();
car.accelerate(); // Output: Car cannot accelerate when stopped.

```

In this example, the `Vehicle` class is an abstract class with a common property `isRunning`, and it defines two methods `start()` and `stop()`. It also has an abstract method `accelerate()`. The `Car` class extends the `Vehicle` class and provides a concrete implementation of the `accelerate()` method. The abstract class provides a partial implementation of the `accelerate()` method by checking if the vehicle is running or stopped before accelerating.

- Establish a Hierarchy: Abstract classes are used to establish an inheritance hierarchy among related classes. They represent a higher-level concept or category that is common to the subclasses. The abstract class defines the common attributes and behavior that are essential to all the subclasses, while the subclasses can further specialize and extend the functionality as per their specific requirements.

```
class Employee {
  constructor(name, salary) {
    this.name = name;
    this.salary = salary;
  }

  // Abstract method
  calculatePay() {
    throw new Error("This method must be implemented.");
  }
}

class FullTimeEmployee extends Employee {
  calculatePay() {
    return this.salary;
  }
}

class PartTimeEmployee extends Employee {
  calculatePay() {
    return this.salary * 0.5;
  }
}

const fullTimeEmployee = new FullTimeEmployee("John Doe", 5000);
console.log(fullTimeEmployee.calculatePay()); // Output: 5000

const partTimeEmployee = new PartTimeEmployee("Jane Smith", 2000);
console.log(partTimeEmployee.calculatePay()); // Output: 1000
```

In this example, the `Employee` class is an abstract class with common properties `name` and `salary`, and it defines an abstract method `calculatePay()`. The `FullTimeEmployee` and `PartTimeEmployee` classes extend the `Employee` class and provide concrete implementations of the `calculatePay()` method. The abstract class establishes a hierarchy by defining the common attributes and behavior that are essential to all employees, while the subclasses specialize and extend the functionality based on their employment type.

- Serve as a Template: Abstract classes can serve as a template or blueprint for creating new classes. They define the structure and requirements that subclasses must fulfill. Developers can derive new classes from the abstract class, providing

concrete implementations for the abstract methods and properties. This allows for consistency and standardization in the creation of related classes.

```
class Shape {
  constructor() {
    if (new.target === Shape) {
      throw new Error("Shape cannot be instantiated directly.");
    }
  }

  // Abstract method
  calculateArea() {
    throw new Error("This method must be implemented.");
  }
}

class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }

  calculateArea() {
    return Math.PI * this.radius * this.radius;
  }
}

class Rectangle extends Shape {
  constructor(width, height) {
    super();
    this.width = width;
    this.height = height;
  }

  calculateArea() {
    return this.width * this.height;
  }
}

const circle = new Circle(5);
console.log(circle.calculateArea()); // Output: 78.53981633974483

const rectangle = new Rectangle(4, 6);
console.log(rectangle.calculateArea()); // Output: 24
```



- It's important to note that abstract classes cannot be directly instantiated. They serve as a foundation for creating concrete classes through inheritance. Abstract classes provide a powerful mechanism for organizing and structuring code in a hierarchical manner, promoting code reuse, and enforcing common behavior among related classes.

## **7. Introduction to DSA (30 minutes)**

**- Provide a brief introduction to data structures and algorithms.**

- Data structures and algorithms are fundamental concepts in computer science and software development. They play a crucial role in solving problems efficiently, optimizing performance, and managing data effectively.

### **Data Structures:**

- A data structure is a way of organizing and storing data in a computer so that it can be accessed and manipulated efficiently. It defines the layout and organization of data elements and provides operations for accessing, inserting, deleting, and modifying the data. Common data structures include arrays, linked lists, stacks, queues, trees, graphs, and hash tables. Each data structure has its own strengths and weaknesses and is suited for specific types of operations and problem-solving scenarios.

### **Algorithms:**

- An algorithm is a step-by-step procedure or a set of rules for solving a specific problem. It is a well-defined sequence of instructions that takes input, performs some computation or manipulation, and produces an output. Algorithms provide a systematic way to solve problems and achieve desired results. They can involve various techniques such as searching, sorting, graph traversal, dynamic programming, and more. Efficient algorithms are essential for optimizing the performance of software applications and solving complex problems within reasonable time constraints.
- The relationship between data structures and algorithms is closely intertwined. Data structures provide a way to organize and store data, while algorithms define the

operations and manipulations performed on that data structure. The choice of an appropriate data structure and algorithm can greatly impact the efficiency, scalability, and correctness of a solution.

- Understanding data structures and algorithms is crucial for software developers as it enables them to design efficient algorithms, select appropriate data structures for specific use cases, optimize code performance, and solve complex problems in various domains such as artificial intelligence, data analysis, web development, and more. Mastery of these concepts helps in writing efficient and scalable code, improving the performance of applications, and enhancing problem-solving skills.

**- Discuss the importance of understanding DSA for efficient programming.**

Understanding data structures and algorithms (DSA) is crucial for efficient programming due to the following reasons:

1. **Problem Solving:** DSA provides a systematic and structured approach to problem-solving. It equips programmers with a toolkit of data structures and algorithms that can be used to analyze problems, devise efficient solutions, and implement them in code. Without a solid understanding of DSA, programmers may struggle to solve complex problems efficiently and effectively.
2. **Performance Optimization:** Efficient algorithms and appropriate data structures can significantly impact the performance of a program. DSA knowledge allows programmers to select the most suitable data structure and algorithm for a given problem, considering factors such as time complexity, space complexity, and input size. This leads to faster execution times, reduced memory usage, and improved scalability of programs.
3. **Code Reusability:** DSA promotes code reusability and modularity. By implementing commonly used data structures and algorithms, programmers can create reusable components that can be easily integrated into different projects. This saves development time and effort, enhances code maintainability, and ensures consistency across applications.
4. **Scalability:** As the size and complexity of data grow, efficient data structures and algorithms become vital for handling large-scale problems. Understanding DSA helps programmers design scalable solutions that can handle increasing data volumes and

computational requirements. It allows for efficient storage, retrieval, and manipulation of data, enabling applications to scale seamlessly.

5. **Problem Domain Mastery:** Many real-world problems have well-established solutions and patterns based on DSA. By understanding DSA, programmers gain insights into common problem-solving techniques and approaches. This knowledge enables them to identify similarities between new problems and existing solutions, adapting and applying appropriate algorithms and data structures to solve them efficiently.
  6. **Technical Interviews:** DSA knowledge is often a crucial requirement in technical interviews for software engineering positions. Employers assess a candidate's problem-solving skills, algorithmic thinking, and ability to optimize code during interviews. A strong understanding of DSA increases the chances of success in interviews and enhances employability in the software industry.
  7. **Collaboration and Code Understanding:** DSA serves as a common language for communication among programmers. When collaborating on projects, understanding commonly used data structures and algorithms allows developers to understand and reason about each other's code effectively. It enables efficient collaboration, code reviews, and troubleshooting.
- In summary, understanding DSA is essential for efficient programming as it empowers programmers to solve problems more effectively, optimize performance, design scalable solutions, promote code reusability, and enhance collaboration. It forms the foundation for developing high-quality software applications and is a fundamental skill for every programmer.
    - Introduce commonly used data structures such as arrays.
  - Here is a list of common data structures used in computer science and programming:
    - Array
    - Linked List
    - Stack
    - Queue
    - Binary Tree
    - Binary Search Tree
    - Heap
    - Hash Table

- Graph
- Trie
- Set
- Hash Map
- Priority Queue
- AVL Tree
- B-Tree
- Red-Black Tree
- Heap
- Graph
- Doubly Linked List
- Circular Linked List

These are just a few examples of common data structures used in programming. Each data structure has its own characteristics, strengths, and use cases, and understanding their properties and operations is essential for efficient programming and problem-solving

## Day 2

1. Introduction to Algorithms and Analysis (30 minutes)
  - Overview of algorithms and their importance in problem-solving
  - Introduction to algorithm analysis and the concept of time complexity
  - Big O notation and its significance in describing algorithm efficiency
2. Basics of Recursion (45 minutes)
  - Definition and characteristics of recursion
  - Recursive vs. iterative approaches
  - Recursive function structure and recursive calls
  - Examples of recursive algorithms (e.g., factorial, Fibonacci)
3. Time Complexity Analysis (1 hour)
  - Definition of time complexity and its importance
  - Analyzing time complexity using recurrence relations
  - Common time complexity classes (e.g.,  $O(1)$ ,  $O(n)$ ,  $O(\log n)$ ,  $O(n^2)$ )
  - Analyzing time complexity of recursive algorithms using recurrence examples
4. Space Complexity and Recursion (30 minutes)
  - Introduction to space complexity and its significance
  - Analyzing space complexity of recursive algorithms
  - Stack memory and recursion depth
  - Examples of space complexity analysis in recursive algorithms
5. Optimizing Recursion and Tail Recursion (45 minutes)
  - Techniques for optimizing recursive algorithms (e.g., memoization, programming)
  - Introduction to tail recursion and its benefits
  - Tail recursion optimization and its impact on space complexity
6. Case Studies and Practical Examples (30 minutes)
  - Solving real-world problems using recursion
  - Analyzing the time and space complexity of practical recursive algorithms
  - Discussion on trade-offs and best practices in recursive algorithm design
7. Conclusion and Recap (15 minutes)
  - Recap of key concepts and topics covered
  - Importance of understanding algorithm and time/space complexity
  - Additional resources for further learning

Introduction to Algorithms and Analysis (30 minutes)

## Topics to Cover

1. Array Methods (Filter and Map)
2. Object
3. Spread Operator
4. Rest Parameters
5. React Basic
6. Setup First React Project – Understanding Basic File Structure of React App
7. Function Vs Class Components
8. JSX

## Array Methods

JavaScript provides numerous array methods to manipulate and work with arrays efficiently. Below are some commonly used array methods along with examples:

1. `push()`: Adds one or more elements to the end of an array and returns the new length of the array.

```
let fruits = ["apple", "banana", "orange"];
fruits.push("grape");
console.log(fruits); // Output: ["apple", "banana", "orange", "grape"]
```

2. `pop()`: Removes the last element from an array and returns that element.

```
let fruits = ["apple", "banana", "orange"];
let lastFruit = fruits.pop();
console.log(fruits); // Output: ["apple", "banana"]
console.log(lastFruit); // Output: "orange"
```

3. `shift()`: Removes the first element from an array and returns that element.

```
let fruits = ["apple", "banana", "orange"];
let firstFruit = fruits.shift();
console.log(fruits); // Output: ["banana", "orange"]
console.log(firstFruit); // Output: "apple"
```

4. `unshift()`: Adds one or more elements to the beginning of an array and returns the new length of the array.

```
let fruits = ["banana", "orange"];
fruits.unshift("apple", "grape");
console.log(fruits); // Output: ["apple", "grape", "banana", "orange"]
```

5. `slice()`: Returns a new array containing a portion of the original array.

```
let fruits = ["apple", "banana", "orange", "grape"];
let citrus = fruits.slice(1, 3);
console.log(citrus); // Output: ["banana", "orange"]
```

6. `splice()`: Adds or removes elements from an array at a specified index.

```
let fruits = ["apple", "banana", "orange", "grape"];
fruits.splice(1, 2, "kiwi", "melon");
console.log(fruits); // Output: ["apple", "kiwi", "melon", "grape"]
```

7. `indexOf()`: Returns the first index at which a given element can be found in an array, or -1 if it is not present.

```
let fruits = ["apple", "banana", "orange"];
let index = fruits.indexOf("banana");
console.log(index); // Output: 1
```

8. `filter()`: Creates a new array with all elements that pass the test implemented by the provided function.

```
let numbers = [1, 2, 3, 4, 5];
let evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Output: [2, 4]
```

9. `map()`: Creates a new array with the results of calling a provided function on every element in the calling array.

```
let numbers = [1, 2, 3];
let squaredNumbers = numbers.map(num => num * num);
console.log(squaredNumbers); // Output: [1, 4, 9]
```

## Objects

In JavaScript, objects are a fundamental data structure that allows you to store and organize data in key-value pairs. An object can hold various data types, including primitive values, arrays, functions, and even other objects. Objects are a cornerstone of JavaScript and are used extensively in the language for data representation and manipulation.

Object Literal Syntax:

The simplest way to create an object in JavaScript is by using the object literal syntax, which uses curly braces `{}`.

```
// Example of an object representing a person
let person = {
  name: "John Doe",
  age: 30,
  city: "New York",
  hobbies: ["reading", "swimming"],
  greet: function() {
    console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
  }
};

console.log(person.name); // Output: "John Doe"
console.log(person.age); // Output: 30
console.log(person.hobbies); // Output: ["reading", "swimming"]
person.greet(); // Output: "Hello, my name is John Doe and I'm 30 years old."
```

### Object Properties:

Object properties are accessed using dot notation or square brackets. Dot notation is simpler and more common, but square brackets are useful when the property name contains special characters or is stored in a variable.

```
let person = {
  name: "John Doe",
  age: 30,
};

console.log(person.name); // Output: "John Doe"
console.log(person["age"]); // Output: 30

let propertyName = "age";
console.log(person[propertyName]); // Output: 30
```

### Adding and Modifying Properties:

You can add or modify properties in an object at any time.

```
let person = {
  name: "John Doe",
  age: 30,
};

person.city = "New York";
person["occupation"] = "Engineer";

console.log(person); // Output: { name: "John Doe", age: 30, city: "New York", occupation: "Engineer" }
```

### Object Methods:



Objects can also contain functions, known as methods.

```
let calculator = {
  add: function(a, b) {
    return a + b;
  },
  subtract: function(a, b) {
    return a - b;
  }
};

console.log(calculator.add(5, 3)); // Output: 8
console.log(calculator.subtract(10, 4)); // Output: 6
```

The 'this' Keyword:

Inside an object method, you can use the `this` keyword to refer to the object itself.

```
let person = {
  name: "John Doe",
  age: 30,
  greet: function() {
    console.log(`Hello, my name is ${this.name}.`);
  }
};

person.greet(); // Output: "Hello, my name is John Doe."
```

Nested Objects:

Objects can contain other objects, allowing you to create complex data structures.

```
let person = {
  name: "John Doe",
  address: {
    city: "New York",
    zipCode: "10001"
  }
};

console.log(person.address.city); // Output: "New York"
```

Loop Over Objects

There are 4 common ways to loop over them

1. For..in
2. Object.keys()
3. Object.values()
4. Object.entries()

## Code Examples

```
let person = {
  name: "John Doe",
  age: 30,
  city: "New York"
};

for (let key in person) {
  console.log(`${key}: ${person[key]}`);
}
```

```
let person = {
  name: "John Doe",
  age: 30,
  city: "New York"
};

let keys = Object.keys(person);
for (let i = 0; i < keys.length; i++) {
  let key = keys[i];
  console.log(`${key}: ${person[key]}`);
}
```

```
let person = {
  name: "John Doe",
  age: 30,
  city: "New York"
};

let values = Object.values(person);
for (let value of values) {
  console.log(value);
}
```

```
let person = {
  name: "John Doe",
  age: 30,
  city: "New York"
};

let entries = Object.entries(person);
for (let [key, value] of entries) {
  console.log(`${key}: ${value}`);
}
```

## Spread Operator

The spread operator in JavaScript is a powerful feature introduced in ES6 (ECMAScript 2015). It is denoted by three dots `...` and allows you to expand an iterable (e.g., arrays, strings, objects) into

individual elements. The spread operator can be used in various scenarios to simplify code and make it more concise.

### 1. Spread Operator with Arrays:

```
// Example 1: Concatenating arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combinedArray = [...arr1, ...arr2];
console.log(combinedArray); // Output: [1, 2, 3, 4, 5, 6]

// Example 2: Cloning an array
const originalArray = [10, 20, 30];
const clonedArray = [...originalArray];
console.log(clonedArray); // Output: [10, 20, 30]

// Example 3: Using spread with array literals
const numbers = [1, 2, 3];
const newArray = [0, ...numbers, 4, 5];
console.log(newArray); // Output: [0, 1, 2, 3, 4, 5]
```

### 2. Spread Operator with Objects:

```
// Example 1: Merging objects
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const mergedObject = { ...obj1, ...obj2 };
console.log(mergedObject); // Output: { a: 1, b: 2, c: 3, d: 4 }

// Example 2: Cloning an object
const originalObject = { x: 10, y: 20 };
const clonedObject = { ...originalObject };
console.log(clonedObject); // Output: { x: 10, y: 20 }

// Example 3: Using spread with object literals (ES9)
const coordinates = { x: 5, ...{ y: 10, z: 15 } };
console.log(coordinates); // Output: { x: 5, y: 10, z: 15 }
```

### 3. Spread Operator with Function Arguments:

```
// Example 1: Passing an array as function arguments
function sum(a, b, c) {
  return a + b + c;
}
const numbers = [1, 2, 3];
const result = sum(...numbers);
console.log(result); // Output: 6

// Example 2: Finding the maximum number in an array
```

```
const numbers = [10, 5, 8];
const maxNumber = Math.max(...numbers);
console.log(maxNumber); // Output: 10
```

#### 4. Spread Operator with Strings:

```
// Example 1: Splitting a string into an array of characters
const str = "Hello";
const charArray = [...str];
console.log(charArray); // Output: ["H", "e", "l", "l", "o"]

// Example 2: Combining strings into a new string
const greeting = "Hello, ";
const name = "John";
const message = greeting + ...name;
console.log(message); // Output: "Hello, John"
```

The spread operator simplifies various coding tasks by providing a concise syntax for combining arrays, objects, strings, and function arguments. It's a versatile feature that significantly improves the readability and maintainability of your JavaScript code.

#### Rest Parameters

Rest parameters in JavaScript allow you to represent an indefinite number of function arguments as an array. It is denoted by three dots `...` followed by a parameter name and can only be used as the last parameter in a function. Rest parameters collect all the remaining arguments passed to a function into an array, which can then be manipulated or accessed within the function body.

```
function sum(...numbers) {
  let total = 0;
  for (let num of numbers) {
    total += num;
  }
  return total;
}

console.log(sum(1, 2)); // Output: 3
console.log(sum(1, 2, 3, 4, 5)); // Output: 15
```

#### React Basics

React is an open-source JavaScript library for building user interfaces. It was developed by Facebook and is widely used for creating web applications, particularly Single Page Applications (SPAs). React follows a component-based architecture, allowing developers to break down the user interface into reusable and manageable pieces called components. Here are the key concepts of React:

## 1. Components:

Components are the building blocks of React applications. A component is a self-contained, reusable piece of user interface that can be composed together to form complex UI structures. React components can be of two types: functional components and class components.

### Functional Components (stateless components):

```
import React from 'react';

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

export default Greeting;
```

### Class

```
import React, { Component } from 'react';

class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return <div>{this.state.count}</div>;
  }
}

export default Counter;
```

## 2. JSX (JavaScript XML):

JSX is a syntax extension for JavaScript that allows you to write HTML-like code in your React components. It makes the code more readable and concise. JSX gets transpiled into regular JavaScript by build tools like Babel.

```
import React from 'react';

function App() {
  return (
    <div>
      <h1>Hello, React!</h1>
    </div>
  );
}
```

```

    <p>This is a JSX example.</p>
  </div>
);
}

export default App;

```

### 3. Props:

Props (short for properties) are used to pass data from parent components to child components. Props are read-only and cannot be modified by the child component.

```

import React from 'react';

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

function App() {
  return <Greeting name="John" />;
}

export default App;

```

### 4. State:

State is used to manage data that can change within a component. Class components have a `state` object, while functional components can use the `useState` hook to introduce state.

```

import React, { useState } from 'react';

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

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default Counter;

```

### 5. Lifecycle Methods:

Class components have lifecycle methods that allow you to perform actions at specific stages of a component's lifecycle. However, with the introduction of React Hooks, many lifecycle methods have been replaced by the `useEffect` hook in functional components.

### 6. Virtual DOM:

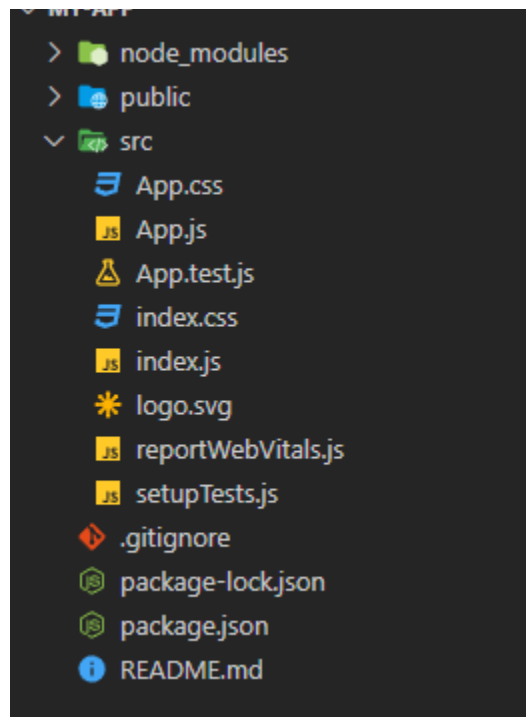
React uses a Virtual DOM to efficiently update the actual DOM. When there are changes to the state or props of a component, React creates a lightweight copy of the Virtual DOM and compares it to the previous one. Only the necessary changes are applied to the actual DOM, making React fast and efficient.

### Setup First React Project

1. Provided you have node installed on your machine
2. Run the following CMD Command **`npx create-react-app app-name`**

```
C:\Users\Shaik Abdul Faraz\Desktop\Learning>npx create-react-app my-app
Creating a new React app in C:\Users\Shaik Abdul Faraz\Desktop\Learning\my-app.
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...
[Progress Bar] \ idealTree:eslint: timing idealTree:node_modules/eslint Completed in 522ms
```

3. Once the project is created , navigate to the project directory and run **`npm start`** , That's it your first react-app will load



1. **Public/:** This directory contains static assets that will be served as-is by the server. The `index.html` file is the main HTML file for the application, and it is the entry point for the React app. The `favicon.ico` is the icon displayed in the browser tab.
2. **src/:** This is the main source directory for the React application.
3. **Components/:** This directory contains reusable components used throughout the application. Components are typically written as functional components or class components.
4. **Pages/:** This directory contains components that represent individual pages/routes of the application. Pages are usually composed of multiple components and are used to define the different views of the application.
5. **App.js:** This is the main component of the application and serves as the root component. It typically holds the routing configuration and renders the various pages and components.
6. **index.js:** This is the entry point for the React application. It is responsible for rendering the `App` component and attaching it to the DOM.
7. **index.css:** This is the main CSS file for the application. You can include additional CSS files or use CSS-in-JS libraries like `styled-components` if needed.
8. **package.json:** This file holds the project configuration and dependencies. It also contains scripts for running, building, and testing the application.
9. **node\_modules/:** This directory contains all the dependencies installed via `npm` or `yarn`. It's automatically generated and should not be manually modified.



```
import logo from "./logo.svg";
import "./App.css";

function App() {
  const user = {
    name: "John Doe",
    age: 30,
  };

  return (
    <div>
      <p>User Name: {user.name}</p>
      <p>User Age: {user.age}</p>
    </div>
  );
}

export default App;
```

## Creating Components – Functional Vs Class

### 1. Functional Component

- a. Create a new Folder named **Greetings**
- b. Create a new JS file inside it named **Greetings.js**
- c. Inside the Greetings.js Put the following Code

```
function Greetings() {
  const greet = {
    msg: "Hello, world"
  };

  return (
    <div>
      {greet.msg}
    </div>
  );
}

export default Greetings;
```

Make sure function Name and Export name matches also the file name

Now in order to re-use it. Go the app component

```
return (
  <div>
    <p>User Name: {user.name}</p>
    <p>User Age: {user.age}</p>
    <Greetings/>
  </div>
);
```

Now the text Highlighted in Green shows that it a re-usable component

Topics to Cover

1. Adding Functions in React Functional Components
2. Data Binding in React
3. Injecting Bootstrap in React
4. What are props in react
5. Class Components in React
6. React Hooks
7. Fetching Data from API in React

### Adding Functions in React Functional Components

You can add a function within the `App` component just like any other JavaScript code. The function can be defined anywhere inside the `App` function body, and it can be used to perform any tasks you need within the component.

```
import logo from "./logo.svg";
import "./App.css";
import Greetings from "./Greetings/Greetings";

function App() {
  const user = {
    name: "John Doe",
    age: 30,
  };

  const calculateBirthYear = () => {
    const currentYear = new Date().getFullYear();
    return currentYear - user.age;
  };

  return (
    <div>
      <p>User Name: {user.name}</p>
      <p>User Age: {user.age}</p>
      <p>User Birth Year: {calculateBirthYear()}</p>
      <Greetings/>
    </div>
  );
}

export default App;
```

For example, let's add a function called `calculateBirthYear` inside the `App` component that calculates the birth year of the user based on their current age:

In this example, the `calculateBirthYear` function takes the current year using `new Date().getFullYear()` and then subtracts the user's age to determine their birth year. The result is displayed in the component using the `{calculateBirthYear()}` expression inside the JSX.

You can add other functions within the `App` component to handle various tasks, such as data manipulation, event handling, or API calls. Keep in mind that the `App` component is just a regular JavaScript function, and you can use JavaScript to define any additional functions you need.

## Data Binding in React

There are two types of data binding in React

1. One-way data Binding
2. Two-way data Binding

### One-way data Binding

One-way data binding means that data flows only in one direction, typically from the component's state to the UI. When the state is updated, the changes are propagated to the UI components that depend on that data, causing them to re-render with the updated values.

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

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

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

In this example, the `count` variable is the state that holds the current count value. When the "Increment" button is clicked, the `increment` function is called, updating the `count` state using `setCount`. As a result, the UI automatically reflects the new value of `count`.

### Two-way data Binding

Two-way data binding allows data to flow both from the component's state to the UI and from the UI back to the state. This is typically used in form elements where the user input should be synchronized with the component's state.

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

function TextInput() {
  const [text, setText] = useState('');
```

```
const handleChange = (event) => {
  setText(event.target.value);
};

return (
  <div>
    <input type="text" value={text} onChange={handleChange} />
    <p>You typed: {text}</p>
  </div>
);
}
```

### Looping over a list of data

In React, you can loop over a list of data and render elements dynamically using the `map` method of JavaScript arrays. The `map` method allows you to iterate through each item in the array, perform some operations on each item, and return a new array with the results. In the context of React, you use `map` to dynamically generate React elements for each item in the data list.

```
import React from 'react';

function DataList() {
  const data = [
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' },
    { id: 3, name: 'Item 3' },
    { id: 4, name: 'Item 4' },
  ];

  return (
    <div>
      <h2>List of Items:</h2>
      <ul>
        {data.map(item => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default DataList;
```

### Injecting Bootstrap in React

In order to inject Bootstrap(CSS Framework) For your react Application follow these two steps

1. **npm install bootstrap**
2. **Open the `src/index.js` file in your project and import Bootstrap's CSS file at the top of the file**

```
import React from 'react';
import ReactDOM from 'react-dom';
import 'bootstrap/dist/css/bootstrap.min.css'; // Import Bootstrap CSS
```

```
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
reportWebVitals();
```

Now you can use Bootstrap components and styles in your React components. For example, you can modify your `App.js` file to include a Bootstrap Navbar:

```
import React from 'react';
import './App.css';

function App() {
  return (
    <div>
      <nav className="navbar navbar-expand-lg navbar-dark bg-dark">
        <a className="navbar-brand" href="/">My App</a>
        <button className="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav"
aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
          <span className="navbar-toggler-icon"></span>
        </button>
        <div className="collapse navbar-collapse" id="navbarNav">
          <ul className="navbar-nav">
            <li className="nav-item active">
              <a className="nav-link" href="/">Home</a>
            </li>
            <li className="nav-item">
              <a className="nav-link" href="/about">About</a>
            </li>
            <li className="nav-item">
              <a className="nav-link" href="/contact">Contact</a>
            </li>
          </ul>
        </div>
      </nav>
      <div className="container">
        { /* Your app content goes here */ }
      </div>
    </div>
  );
}

export default App;
```

### Props in React

In React, props (short for properties) are used to pass data from a parent component to a child component. They allow you to pass data down the component tree and make your components more flexible and reusable.

Step by step process to pass data from parent to a child

We are going to follow two steps

1. Configuration of props in parent
2. Configuration of props in child

### Configuration of props in parent

Render the child component in parent using traditional syntax and pass data as follows

```
<Greetings name="John Doe" age={30} />
<Greetings name="Jane Smith" age={25} />
```

Now the next step is that we want the child to receive these props

### Configuration of props in child

```
const Greetings = (props) => {
  return (
    <div>
      {props.name}
      {props.age}
    </div>
  );
}

export default Greetings;
```

This will print the output

```
John Doe30
Jane Smith25
```

### Class Components in React

We are going to create an increment counter to understand the class component and its features

## Class Component

Count: 9

Increment

```
import React from 'react';

class ClassComponent extends React.Component {
  constructor(props) {
    super(props);
    // You can initialize the component's state in the constructor
    this.state = {
```

```

    count: 0
  };
}

// You can also define custom methods for the component
incrementCount() {
  this.setState((prevState) => ({
    count: prevState.count + 1
  }));
}

render() {
  return (
    <div>
      <h1>Class Component</h1>
      <p>Count: {this.state.count}</p>
      <button onClick={() => this.incrementCount()}>Increment</button>
    </div>
  );
}
}

export default ClassComponent;

```

You can use the CLI command to generate class component

Aspect	Class Components	Functional Components with Hooks
Definition	Defined as JavaScript classes	Defined as JavaScript functions
State Management	Uses <code>this.state</code> and <code>this.setState</code> for state	Uses <code>useState</code> hook for state
Lifecycle Methods	Has lifecycle methods like <code>componentDidMount</code> , <code>componentDidUpdate</code> , etc.	Uses <code>useEffect</code> hook for side effects and lifecycle tasks
Readability	Can be more verbose with lifecycle methods	More concise and easier to read
Reusability	Can use inheritance for code sharing	Emphasizes composition and code sharing
Performance Optimization	Can use <code>shouldComponentUpdate</code> for optimization	Can use <code>React.memo</code> and <code>useMemo</code> hooks
Syntax and Boilerplate	Requires <code>constructor</code> and <code>render</code> methods	Eliminates <code>constructor</code> and <code>render</code> methods
State Initialization	State initialized in <code>constructor</code>	State initialized using <code>useState</code> hook
Example	See the class component example above	See the functional component example above

## React Hooks

Hooks in React are functions that allow you to use state and other React features in functional components without the need for class components. They were introduced in React 16.8 to simplify state management and lifecycle-related logic in functional components, making them more powerful and expressive.

There are several built-in hooks provided by React, and you can also create custom hooks to encapsulate reusable logic in your applications.

Some of the most commonly used built-in hooks are:

1. `useState`: Used to add state to functional components. It returns an array with the current state value and a function to update that state.

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

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

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

2. `useEffect`: Used to perform side effects in functional components, such as data fetching, subscriptions, or updating the DOM. It's similar to `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` lifecycle methods in class components.

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

function DataFetching() {
  const [data, setData] = useState([]);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data))
      .catch(error => console.error(error));
  }, []);

  return (
    <div>
      <h2>Data from API:</h2>
      <ul>
```



```

    {data.map(item => (
      <li key={item.id}>{item.name}</li>
    ))}
  </ul>
</div>
);
}

```

3. `useContext`: Used to access data from a React context in functional components.
4. `useReducer`: An alternative to `useState`, mainly used for more complex state management.
5. `useCallback`: Used to memoize functions to prevent unnecessary re-renders.
6. `useMemo`: Used to memoize values to prevent unnecessary recalculations.
7. `useRef`: Used to create mutable values that persist across re-renders.

Using hooks, you can build more maintainable and organized components by keeping related logic together and avoiding the complexities of class components. Hooks also enable better performance optimizations, as they make it easier to identify and control the scope of state and side effects.

It's important to note that hooks can only be used in functional components and should not be used inside loops, conditions, or nested functions. Also, make sure you follow the rules of hooks to avoid any issues with their behavior.

### Fetching Data from API in React

Create a new component DataFetching

```

import React, { useState, useEffect } from 'react';

function DataFetching() {
  const [data, setData] = useState([]);

  useEffect(() => {
    fetch('https://fakestoreapi.com/products')
      .then(response => response.json())
      .then(data => setData(data))
      .catch(error => console.error(error));
  }, []);

  return (
    <div>
      <h2>Data from API:</h2>
      <ul>
        {data.map(item => (
          <li key={item.id}>{item.title}</li>

```

```

    })}
  </ul>
</div>
);
}

export default DataFetching;

```

and render this in your app.js component

## Topics to Cover

1. Promises
2. Async Await
3. React Router
4. Creating Registration Form
5. Performing Front End Validation
6. Creating Login Form
7. Performing Front End Validation

## Promises

Promises in JS are very similar to promises in real life. If you make a promise to someone for something. There can be two possibilities

1. Full fill the promise
2. Not Full fill the promise

These terms in JS are called **resolve** for Full-filling and **reject** for not full-filling it.

In order to create a promise , create an object or Promise as follows

```

const x = new Promise((resolve, reject) =>{
})

```

Now in this method write some logic to either resolve a promise or reject it

```

const x = new Promise((resolve, reject) =>{
  let a = 2;
  if(a ===2) {
    resolve('Success')
  } else {
    reject('Failed')
  }
})

```

Now you need to call this promise using both **then** and **catch** block.

**then** → **resolve** and **catch** → **reject**

```

x.then((message) => {

```

```

    console.log(message);
  }).catch((err) => {
    console.log(err);
  })

```

If the promise is resolved the **then** block will be invoked , if it is rejected → **catch** block will be executed

In our case we will get **'Success'**

If you have multiple promises and want to run logic if all of them succeed then we are going to use **all** method

```

const x = new Promise((resolve, reject) =>{
  let a = 2;
  if(a ===2) {
    resolve('Success')
  } else {
    reject('Failed')
  }
})

const y = new Promise((resolve, reject) =>{
  let a = 2;
  if(a ===2) {
    resolve('Success')
  } else {
    reject('Failed')
  }
})

const z = new Promise((resolve, reject) =>{
  let a = 2;
  if(a ===2) {
    resolve('Success')
  } else {
    reject('Failed')
  }
})

Promise.all([x,y,z]).then((message) => {
  console.log(message);
})

```

The then block will only be invoked only if all the promises x,y and z are **resolved** , otherwise it will catch the error

### Async Await

Async/await is a feature in JavaScript introduced with ECMAScript 2017 (ES8) that provides a more elegant way to work with asynchronous code. It's built on top of Promises and makes asynchronous operations look and feel more like synchronous code, making it easier to read and write

```

// Function to fetch user data from the API
async function fetchUserData(userId) {
  const apiUrl = `https://jsonplaceholder.typicode.com/users/${userId}`;

  try {

```

```

const response = await fetch(apiUrl);

if (!response.ok) {
  throw new Error('Network response was not ok.');
```

```

}

const userData = await response.json();
return userData;
} catch (error) {
  console.error('Error fetching user data:', error.message);
  return null;
}
}

// Using the fetchUserData function with async/await
async function getUserAndDisplay(userId) {
  console.log('LOADING....');
  const userData = await fetchUserData(userId);

  if (userData) {
    console.log('User data:', userData);
  } else {
    console.log('User not found.');
```

```

  }
}

// Call the function with a user ID (e.g., 1)
getUserAndDisplay(1);

```

## React Router

React Router is a popular library for handling routing in React applications. It allows you to create a single-page application (SPA) with multiple pages that are dynamically rendered without requiring a full page refresh. React Router enables you to define routes, map them to different components, and handle navigation between these components.

Key features of React Router:

1. **Declarative routing:** React Router uses a declarative approach to define routes in your application. You define the routes and their corresponding components using JSX syntax.
2. **Nested routes:** React Router supports nested routes, meaning you can have routes within routes, allowing you to create complex page structures.
3. **URL Parameters:** You can pass parameters within the URL to your components using React Router. These parameters can be accessed within the component to display dynamic content.
4. **Route Matching:** React Router employs a flexible and powerful matching algorithm to determine which route should be rendered based on the current URL.

5. History Management: React Router manages browser history, allowing you to navigate through your application using browser history back and forward buttons.

6. BrowserRouter and HashRouter: React Router provides two main router types: `BrowserRouter` and `HashRouter`. The `BrowserRouter` uses the HTML5 history API for clean URLs (e.g., `/home`, `/about`). The `HashRouter` uses the hash portion of the URL (e.g., `#/home`, `#/about`) and is useful for hosting on static servers or environments where server-side configuration is not possible.

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

// Components for different pages
const Home = () => <div>Home Page</div>;
const About = () => <div>About Page</div>;
const Contact = () => <div>Contact Page</div>;

const App = () => {
  return (
    <Router>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/about">About</Link>
          </li>
          <li>
            <Link to="/contact">Contact</Link>
          </li>
        </ul>
      </nav>

      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </Router>
  );
};

export default App;
```

In this example, we import the necessary components from `react-router-dom`, including `BrowserRouter`, `Route`, and `Link`. We define three components (`Home`, `About`, and `Contact`) to represent different pages of our application. Inside the `App` component, we use `Router` to wrap our navigation (`Link` components) and `Route` components to map each page component to a specific URL path.

When the user clicks on a navigation link, React Router will handle the routing and render the appropriate component without a full page reload, resulting in a smooth single-page application experience.

### Creating React App – Login , Registration with Routers

```
import Register from "../Components/Register";
import Login from "../Components/Login";
import Nav from "../Components/Nav";
import "../Components/App.css";
import React from "react";
import { BrowserRouter as Router, Switch, Route } from "react-router-dom";
import { Link } from "react-router-dom/cjs/react-router-dom.min";

export default function App() {
  return (
    <Router>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/login">login</Link>
          </li>
          <li>
            <Link to="/register">register</Link>
          </li>
        </ul>
      </nav>
      <Switch>
        <Route path="/" exact component={Home} />

        <Route path="/register" component={Register} />

        <Route path="/login" component={Login} />
      </Switch>
    </Router>
  );
}

const Home = () => (
  <div>
    <h1>Home Page</h1>
  </div>
);
```

### Register Component

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

const RegistrationForm = () => {
  const [formData, setFormData] = useState({
    firstName: '',
    lastName: '',
```

```

    email: '',
    password: '',
    confirmPassword: '',
  });

  const [errors, setErrors] = useState({});

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData((prevData) => ({
      ...prevData,
      [name]: value,
    }));
  };

  const handleSubmit = (e) => {
    e.preventDefault();

    // Basic validation
    const newErrors = {};
    if (!formData.firstName) {
      newErrors.firstName = 'First Name is required.';
    }
    if (!formData.lastName) {
      newErrors.lastName = 'Last Name is required.';
    }
    if (!formData.email) {
      newErrors.email = 'Email is required.';
    }
    if (!formData.password) {
      newErrors.password = 'Password is required.';
    } else if (formData.password.length < 6) {
      newErrors.password = 'Password must be at least 6 characters.';
    }
    if (formData.password !== formData.confirmPassword) {
      newErrors.confirmPassword = 'Passwords do not match.';
    }

    setErrors(newErrors);

    // If no errors, proceed with registration logic here (e.g., send data to the server)
    if (Object.keys(newErrors).length === 0) {
      console.log(formData);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label htmlFor="firstName">First Name</label>
        <input
          type="text"
          id="firstName"
          name="firstName"
          value={formData.firstName}
          onChange={handleChange}
        />
        {errors.firstName && <div className="error">{errors.firstName}</div>}
      </div>
    </form>
  );

```

```

    </div>
    <div>
      <label htmlFor="lastName">Last Name</label>
      <input
        type="text"
        id="lastName"
        name="lastName"
        value={formData.lastName}
        onChange={handleChange}
      />
      {errors.lastName && <div className="error">{errors.lastName}</div>}
    </div>
    <div>
      <label htmlFor="email">Email</label>
      <input
        type="email"
        id="email"
        name="email"
        value={formData.email}
        onChange={handleChange}
      />
      {errors.email && <div className="error">{errors.email}</div>}
    </div>
    <div>
      <label htmlFor="password">Password</label>
      <input
        type="password"
        id="password"
        name="password"
        value={formData.password}
        onChange={handleChange}
      />
      {errors.password && <div className="error">{errors.password}</div>}
    </div>
    <div>
      <label htmlFor="confirmPassword">Confirm Password</label>
      <input
        type="password"
        id="confirmPassword"
        name="confirmPassword"
        value={formData.confirmPassword}
        onChange={handleChange}
      />
      {errors.confirmPassword && (
        <div className="error">{errors.confirmPassword}</div>
      )}
    </div>
    <button type="submit">Register</button>
  </form>
);
};

export default RegistrationForm;

```

## Login Component



```

import React, { useState } from 'react';

const LoginForm = () => {
  const [loginData, setLoginData] = useState({ email: '', password: '' });
  const [errors, setErrors] = useState({ email: '', password: '' });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setLoginData((prevData) => ({
      ...prevData,
      [name]: value,
    }));
  };

  const handleSubmit = (e) => {
    e.preventDefault();

    // Basic validation
    const newErrors = {};
    if (!loginData.email) {
      newErrors.email = 'Email is required.';
    }
    if (!loginData.password) {
      newErrors.password = 'Password is required.';
    }

    setErrors(newErrors);

    // If no errors, proceed with login logic here (e.g., send data to the server)
    if (Object.keys(newErrors).length === 0) {
      console.log(loginData);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label htmlFor="email">Email</label>
        <input
          type="email"
          id="email"
          name="email"
          value={loginData.email}
          onChange={handleChange}
        />
        {errors.email && <div className="error">{errors.email}</div>}
      </div>
      <div>
        <label htmlFor="password">Password</label>
        <input
          type="password"
          id="password"
          name="password"
          value={loginData.password}
          onChange={handleChange}
        />
        {errors.password && <div className="error">{errors.password}</div>}
      </div>
    </form>
  );
};

```

```

    <button type="submit">Login</button>
  </form>
);
};

export default LoginForm;

```

## App CSS

```

/* Add this CSS in your component or a separate CSS file */
form {
  width: 300px;
  margin: 0 auto;
  padding: 20px;
  border: 1px solid #ccc;
  border-radius: 5px;
}

div {
  margin-bottom: 15px;
}

label {
  display: block;
  font-weight: bold;
}

input {
  width: 100%;
  padding: 8px;
  border: 1px solid #ccc;
  border-radius: 5px;
}

.error {
  color: red;
  font-size: 12px;
  margin-top: 5px;
}

button {
  width: 100%;
  padding: 10px;
  background-color: #007bff;
  color: #fff;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

button:hover {
  background-color: #0056b3;
}

/* Add this CSS in your component or a separate CSS file */
form {
  width: 300px;

```

```
margin: 0 auto;
padding: 20px;
border: 1px solid #ccc;
border-radius: 5px;
}

div {
  margin-bottom: 15px;
}

label {
  display: block;
  font-weight: bold;
}

input {
  width: 100%;
  padding: 8px;
  border: 1px solid #ccc;
  border-radius: 5px;
}

.error {
  color: red;
  font-size: 12px;
  margin-top: 5px;
}

button {
  width: 100%;
  padding: 10px;
  background-color: #007bff;
  color: #fff;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

button:hover {
  background-color: #0056b3;
}
```

You can clone the entire code repo at <https://github.com/12Faraz/LoginRegistration.git>

1. Class Components and State Management
2. React Router
3. Async Await and FetchAPI in React
4. Node Express Theory
5. HTTP request Theory
6. Node Express Project Setup Step by Step

7. Run Node app --> on local host
8. Postman Basics
9. Connect Node to React
  - a. Understanding Developer vs. production Server

### **Class Components and State Management**

Class components and state management are concepts in React that help in building dynamic and interactive user interfaces. Let's break down these concepts in a simplified manner:

1. **Class Components:** Imagine you have a box that can perform different actions. A class component in React is like that box. It's a special type of component that you can use to define more complex functionality and behavior.
2. **State:** Now, think of a special compartment inside the box where you can store and keep track of information that can change over time. This compartment is called "state" in React. It allows you to store data and manage the dynamic aspects of your component.
3. **State Management:** State management is about handling and updating the data stored in the state compartment of your component. You can think of it as taking care of the information inside the compartment, like adding, removing, or modifying items.

In class components, you can define and manage state using a special property called ``state`` that belongs to the component. This ``state`` property holds the data that can change over time. Whenever the state is updated, React will automatically re-render the component to reflect the changes in the user interface.

In this example, we have a class component called Counter. It has a ``state`` property initialized in the constructor with an initial count value of 0.

The ``incrementCount`` method is used to update the count value in the state by calling ``this.setState()``. React will then re-render the component, reflecting the updated count value in the user interface.

The ``render`` method displays the current count value from the state in a paragraph element, along with a button that triggers the ``incrementCount`` method when clicked.

This is just a basic example of state management in class components. It showcases how you can use state to track and update data in your components, enabling them to respond dynamically to user interactions.

Note: React also provides functional components and hooks as an alternative to class components for managing state. Hooks, like `useState`, offer a simpler and more concise way to handle state in functional components.

```
import React, { Component } from 'react';

class Increment extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  incrementCount() {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.incrementCount()}>Increment</button>
      </div>
    );
  }
}

export default Increment;
```

## React Router

React Router is a popular library for handling routing in React applications. It allows you to define routes, map them to specific components, and navigate between different pages or views without triggering a full page reload.

In order to create a React Router , Follow these steps

1. Wrap the entire Component HTML code under **<Router> </Router>**
2. Create link tags pointing to that page you want to navigate
3. Create **<Route>** Tags with a **component** property

Here is the basic Tree structure of a react Router

```
- Router
  - div
    - nav
      - ul
        - li
          - Link (to="/")
            - Home
        - li
          - Link (to="/about")
            - About
        - li
          - Link (to="/contact")
            - Contact
      - Route (exact path="/", component=Home)
        - Home component
      - Route (path="/about", component=About)
        - About component
      - Route (path="/contact", component=Contact)
        - Contact component
```

### Async Await and FetchAPI in React

Async await is a modern JavaScript feature that simplifies working with promises. It allows you to write asynchronous code that looks and behaves more like synchronous code, making it easier to reason about and maintain. It's based on Promises, and any function declared with the `async` keyword automatically returns a Promise.

```
async function test() {
  try {
    const response = await fetch('https://fakestoreapi.com/products');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
```

```

        console.error(error);
    }
}

test();

```

Now this function is just printing the data return in the form of Promise. What we need to do is **return** instead of printing . so I will modify the code as

```

const fetchData = async () => {
  try {
    const response = await fetch("https://fakestoreapi.com/products");
    if (!response.ok) {
      throw new Error("Network response was not ok");
    }
    const jsonData = await response.json();
    setData(jsonData);
  } catch (error) {
    console.error(error);
  }
};

```

```

useEffect(() => {
  fetchData();
}, []);

```

### Why useEffect ?

In the code you provided, the **useEffect** hook is used to execute the **fetchData** function when the component is first rendered.

The **useEffect** hook is a built-in hook in React that allows you to perform side effects in function components. It takes two arguments: a callback function and an optional dependency array.

In this case, the **useEffect** hook is used with an empty dependency array []. This means that the effect will only be executed once when the component is initially rendered.

By using **useEffect** with an empty dependency array, we ensure that the **fetchData** function is only called once when the component is mounted. This is a common pattern for fetching data from APIs or performing other one-time setup tasks in React components.

Without **useEffect**, the **fetchData** function would be called every time the component re-renders, which can cause unnecessary API calls and potential performance issues.

In summary, the **useEffect** hook with an empty dependency array allows us to fetch the data once when the component is mounted, ensuring that the data is retrieved and stored in the component's state only when necessary.

You can remove **useEffect** and call the method on click of a button

And in the template

```
<div className="container">
  {data && (
    <ul>
      {data.map((item) => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  )}
</div>
```

To Understand this code better , let's understand **conditional rendering in React**

There are mostly 3 ways to do this

1. Using IF condition
2. Using Ternary Operator
3. Using Logical AND operator (&&)

**Using IF**

```
import React from 'react';

const MyComponent = ({ isLoggedIn }) => {
  if (isLoggedIn) {
    return <div>Welcome, user!</div>;
  } else {
    return <div>Please log in to continue.</div>;
  }
};

export default MyComponent;
```

**Using Ternary Operator**

```
import React from 'react';
```



```
const MyComponent = ({ isLoggedIn }) => {
  return isLoggedIn ? <div>Welcome, user!</div> : <div>Please log in to
continue.</div>;
};

export default MyComponent;
```

#### Using Logical AND

```
import React from 'react';

const MyComponent = ({ isLoggedIn }) => {
  return isLoggedIn && <div>Welcome, user!</div>;
};

export default MyComponent;
```

#### Node Express Thoery

1. **Node.js:** Node.js is a JavaScript runtime that allows you to run JavaScript code on the server-side. It provides an event-driven, non-blocking I/O model, making it efficient for building scalable network applications.
2. **Express:** Express is a web application framework for Node.js. It provides a set of features and utilities to build web applications and APIs easily. Express simplifies tasks such as routing, request handling, and middleware integration.

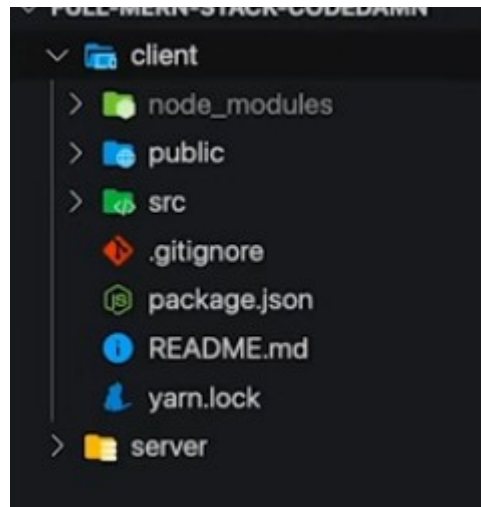
#### Complete Setup for MERN Stack Application – User Authentication PART 1

1. Create a new folder named MERN-Stack

Development = Node.js + react Server

Production = Node js serve +Static React Files

2. Inside main Folder create two more folders
  - a. **Client** → Create app with name client → command → `npx create-react-app client`
  - b. **Server** → Create one index js file inside this folder



3. Now navigate to the server folder and run the command **npm init or yarn init**
4. Now add the express js framework → run the command **npm add express or yarn add express** and verify this in package json file of server folder
5. Now add a new package called **nodemon** in server folder using **npm add nodemon or yarn add nodemone**
6. Confirm then in the package json file

```
},  
  "dependencies": {  
    "express": "^4.17.1",  
    "nodemon": "^2.0.12"  
  }  
}
```

7. Also create a new script in this package json file as follows

```
"scripts": {  
  "dev": "nodemon index.js"  
},
```

8. Next step is creating a new express server , go to index.js file. Every basic node and express app would have 3 basic parts in it

Import Required Packages

HTTP Requests

app listen on a port

For now add this code in index.js

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

app.get('/hello', (req, res) => {
  res.send('hello world')
})

app.listen(1337, () => {
  console.log('Server started on 1337')
})
```

To run this backend code → run the command → **npm dev or npm run dev**

Note that your node app will listen on the port 1337 → **localhost:1337**

Now we have two servers running

1. React Server → localhost:3000
  2. Node js server → localhost:1337
9. Let's create a Basic Registration Form → For saving time visit this github link <https://github.com/12Faraz/LoginRegistration.git> and clone or download the repository
10. Now in the React code , on click of register button I want to call a method , which is already written in the github repository
11. And Once that is done , we will hitting the backend endpoint using the port number mentioned in the node app that is **1337** as follows

```
// If no errors, proceed with registration logic here (e.g., send data to the server)
```

```

if (Object.keys(newErrors).length === 0) {

  const response = await fetch('http://localhost:1337/api/register',
  {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      formData
    })
  })
  )
  const data = await response.json();
  console.log(data);
}

```

Now as we don't have this endpoint in the node app let's create it as follows

```

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

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

app.post('/api/register', (req, res) => {
  console.log(req.body);
  res.json({status:'ok'})
})

app.listen(1337, ()=>{
  console.log('server started')
})

```

We added a new HTTP request POST which will send user data to be stored in the DB to the backend from UI

If we hit this we are going to get an error “**CORS Issues**” to fix this we will be adding a Middleware called cors

To do that , terminate the backend server and run **npm add cors**

Once done , import it in our index.js file and use it as follows

```

const cors = require('cors');

```

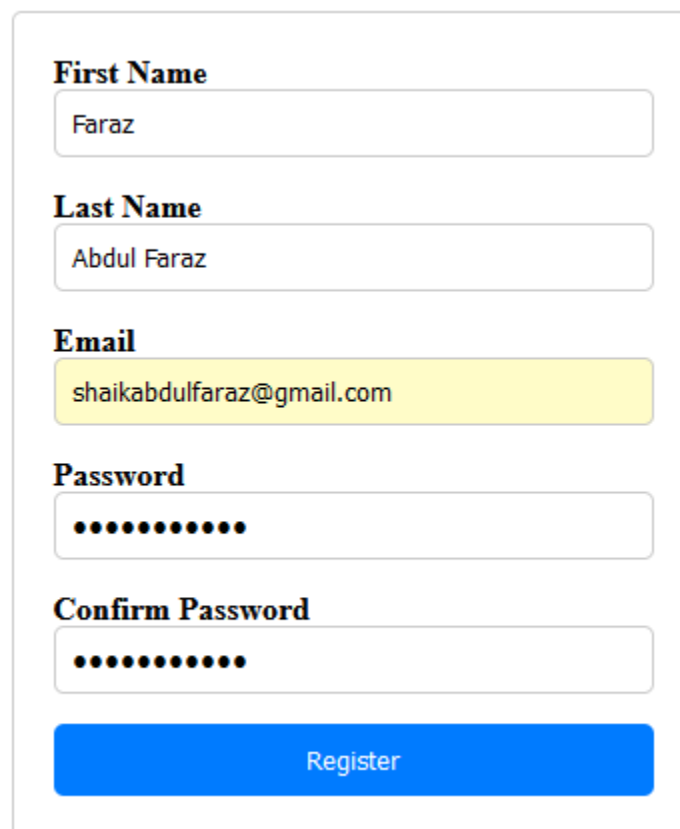
```
app.use(cors());
```

Now when you fill the form and hit the register button → this will return 200 ok response in network tab but you won't be seeing data being console'd you will see **undefined** to fix this problem add this line after app use cors line

```
app.use(express.json());
```

`app.use(express.json())` is a middleware function in Express that is used to parse incoming request bodies with JSON payloads. It allows your Express application to handle JSON data sent by clients in the request body and automatically converts it into a JavaScript object that you can then use in your route handlers.

Now if you click on registration button



The image shows a registration form with the following fields and values:

- First Name:** Faraz
- Last Name:** Abdul Faraz
- Email:** shaikabdufaraz@gmail.com
- Password:** (masked with dots)
- Confirm Password:** (masked with dots)
- Register Button:** A blue button labeled "Register".

And in the console of backend app

```
Server started
{
  formData: {
    firstName: 'Faraz',
    lastName: 'Abdul Faraz',
    email: 'shaikabdufaraz@gmail.com',
    password: 'Abdul Faraz',
    confirmPassword: 'Abdul Faraz'
  }
}
```

Now , let's connect to database → mongodb using a package named **mongoose**

**npm add mongoose**

```
const mongoose = require('mongoose');
```

Now once this is done you need to establish a connection between Node Express App and MongoDB

To do that use **mongoose . connect ()**

In this string you need give in certain format as follows

```
'mongodb://username:password@host:port/databaseName'
```

Before you give that string you need to have a mongo DB account → let's create one

Fill the registration form

#### Email Address

We recommend using your work email

femam786@gmail.com



#### First Name

Shaik Abdul



#### Last Name

Faraz - Trainer



#### Password

••••••••



Your password must:

- Contain at least **8 characters**
- Contain unique characters, numbers, or symbols
- Not contain your email address

#### Company Name

MCEME



☒ I accept the [Privacy Policy](#) and the [Terms of Service](#)



I'm not a robot



reCAPTCHA  
[Privacy](#) - [Terms](#)

Sign up

And make sure you have access to your gmail ID and verify your Email , select this option



## Deploy your database

Use a template below or set up [advanced configuration options](#). You can also edit these configuration options once the cluster is created.

**M10****\$0.08/hour**

For production applications with sophisticated workload requirements.

STORAGE	RAM	vCPU
<b>10 GB</b>	<b>2 GB</b>	<b>2 vCPUs</b>

+ Backups, elastic scalability, & more.

**SERVERLESS****\$0.10/1M reads**

For application development and testing, or workloads with variable traffic.

STORAGE	RAM	vCPU
<b>Up to 1 TB</b>	<b>Auto-scale</b>	<b>Auto-scale</b>

+ Backups, elastic scalability, & more.

**M0****FREE**

For learning and exploring MongoDB in a cloud environment.

STORAGE	RAM	vCPU
<b>512 MB</b>	<b>Shared</b>	<b>Shared</b>

[Compare Features](#)

Provider



Region

★ Recommended region ⓘ

FREE

Create

Free forever! Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

[I'll deploy my database later](#)

[Access Advanced Configuration](#)

Now give a user name and password





## 1 How would you like to authenticate your connection?

Your first user will have permission to read and write any data in your project.

Username and Password

Certificate

 We autogenerated a username and password for your first database user in this project using your MongoDB Cloud registration information. 

Create a database user using a username and password. Users will be given the *read and write to any database* [privilege](#) by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password.

Username

femam786

Password 

E0NFeWt9qmUkCCNs

 Autogenerate Secure Password

 Copy

Create User

Copy this password and save somewhere like notepad

## Add entries to your IP Access List

Only an IP address you add to your Access List will be able to connect to your project's clusters. You can manage existing IP entries via the [Network Access Page](#).

IP Address

Description

Enter IP Address

Enter description

Add My Current IP Address

Add Entry

IP Access List

Description

49.204.20.124/32

My IP Address

 EDIT

 REMOVE

If you want to add any other IP or a Localhost that can access our DB through API then add that API (whitelist) it

Now click on Finish and Close , now you can see this dashboard screen

The screenshot shows the MongoDB Atlas 'Database Deployments' page for 'Project 0'. The left sidebar contains navigation links for 'DEPLOYMENT' (Database, Data Lake), 'SERVICES' (Device Sync, Triggers, Data API, Data Federation, Search, Stream Processing), and 'SECURITY' (Backup, Database Access, Network Access, Advanced). The main content area is titled 'Database Deployments' and includes a search bar, a '+ Create' button, and a 'Load sample datasets to Cluster0.' section with 'Load sample dataset' and 'Dismiss' buttons. Below this is a 'Cluster0' section with 'Connect', 'View Monitoring', and 'Browse Collections' buttons, along with 'FREE' and 'SHARED' labels. The 'Visualize Your Data' section provides a summary of database metrics: Read (R) 0, Write (W) 0, Connections 0, In 0.0 B/s, Out 0.0 B/s, and Data Size 0.0 B / 512.0 MB (8%). It also includes 'Dismiss' and 'Explore Charts' buttons. At the bottom, a table lists deployment details: Version 6.0.8, Region AWS / N. Virginia (us-east-1), Cluster Tier M0 Sandbox (General), Type Replica Set - 3 nodes, Backups Inactive, Linked App Services Loading data..., Atlas SQL Connect, and Atlas Search Create Index. An '+ Add Tag' button is located at the bottom left of the table.

Now Click on

This close-up view focuses on the 'Cluster0' section of the dashboard. It features a row of buttons: 'Cluster0' (with a green dot icon), 'Connect', 'View Monitoring', 'Browse Collections', and a three-dot menu. Below this, the 'Visualize Your Data' section is shown, which includes a description: 'Build dashboards and charts, and embed them in your apps with MongoDB Charts.' It also contains 'Dismiss' and 'Explore Charts' buttons. To the right, a summary of metrics is displayed: Read (R) 0, Write (W) 0, Last 42 seconds, 100.0/s, Connections 0, Last 42 seconds, 100.0, In 0.0 B/s, Out 0.0 B/s, Last 42 seconds, 100.0 B/s, and Data Size 0.0 B / 512.0 MB (8%). An information icon (i) is visible next to the 'Connections' metric.

Connect Button and

## Connect to Cluster0 ✕



### Connect to your application



#### Drivers

Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.)



### Access your data through tools



#### Compass

Explore, modify, and visualize your data with MongoDB's GUI



#### Shell

Quickly add & update data using MongoDB's Javascript command-line interface



#### MongoDB for VS Code

Work with your data in MongoDB directly from your VS Code environment



#### Atlas SQL

Easily connect SQL tools to Atlas for data analysis and visualization



Go Back

Close

Select Compass ,and copy this string

## Connect to Cluster0



### Connecting with MongoDB Compass

I don't have MongoDB Compass installed

I have MongoDB Compass installed

#### 1. Select your operating system and download MongoDB Compass

macOS arm64 (M1) (11.0+)

Download Compass (1.39.0)

or

Copy download URL

Compass is an interactive tool for querying, optimizing, and analyzing your MongoDB data.

#### 2. Copy the connection string, then open MongoDB Compass

mongodb+srv://femam786:<password>@cluster0.p7u6nwc.mongodb.net/

Replace **<password>** with the password for the **femam786** user.

When entering your password, make sure that any special characters are [URL encoded](#).

#### RESOURCES

[Connect with Compass](#)

[Access your Database Users](#)

[Import and Export Data](#)

[Troubleshoot Connections](#)

Go Back

Close

Alright! Let's go to our node app now

```
mongoose
  .connect(
    "mongodb+srv://femam786:8VDv9Sb1c7Udy8Q9@cluster0.p7u6nwc.mongodb.net/users"
  )
  .then(() => {
    console.log("Connected to MongoDB");
  })
  .catch((err) => {
```

```
console.log(err + "Failed to connect to MongoDB");
});
```

Let's understand what I kept in mongoose connect method

`"mongodb+srv://femam786:8VDv9Sb1c7Udy8Q9@cluster0.p7u6nwc.mongodb.net/users"`


1 How would you like to authenticate your connection?



Your first user will have permission to read and write any data in your project.

Username and Password Certificate

**DB Name - Could be any for now**

**Username**  
femam786

**Password** 

 Autogenerate Secure Password  Copy

Create User

Create a database user using a username and password. Users will be given the *read and write to any database* privilege by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password.

We autogenerated a username and password for your first database user in this project using your MongoDB Cloud registration information.

The moment you write our code and run the node app again it should show this in the terminal console

```
Debugger listening on ws://127.0.0.1:50914/ba4e01ec-12f1-426c-9b3e-2daa7965ea30
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Debugger listening on ws://127.0.0.1:50922/8123da3e-8e0f-4b04-b076-4a50f51734c8
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
server started
Connected to MongoDB
█
```

That's it we are now connect to our **cloud** database

Now let us Register the User

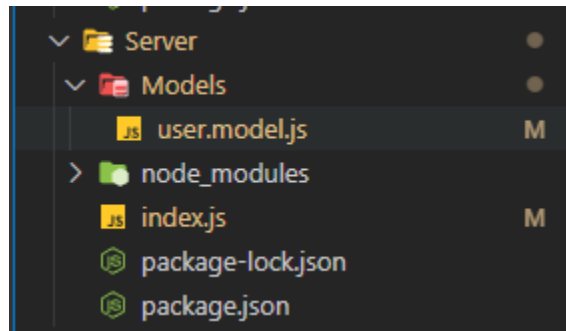
To do so we are going to follow certain steps

1. Create a Model for the user Data that will come from UI → React
2. Import that model in our App.js
3. Create a new end Point

In the indicated Section we are going to write the code to Register the New users.

**1. Create a Model for the user Data that will come from UI → React**

Before we do that create a new folder called **Models** and create a new file in it called **user.model.js** and it could be any name



In this model we are going to create a mongoose Schema (can be understood as a Database Schema or table schema)

```
const mongoose = require("mongoose");

const User = new mongoose.Schema({
  firstName: { type: "string", required: true },
  lastName: { type: "string", required: true },
  email: { type: "string", required: true },
  password: { type: "string", required: true }
},
{ collection: 'user-data'
});

module.exports = mongoose.model('User', User);
```

You see a collection object key which indicates a **collection name** can be understood as **table** in database with that name , required true mean if you don't pass that value the API wont Add data to the Database , also let's make email unique as follows

```
email: { type: "string", required: true , unique: true },
```

## 2. Import that model in our App.js

```
3. const User = require("../Models/user.model");
```

## 3. Create a New End point

```
app.post("/api/register", (req, res) => {  
});
```

Our App.js file looks like this

```
const express = require("express");  
const app = express();  
const cors = require("cors");  
const mongoose = require("mongoose");  
const User = require("../Models/user.model");  
app.use(cors());  
app.use(express.json());  
mongoose  
  .connect(  
    "mongodb+srv://femam786:8VDv9Sb1c7Udy8Q9@cluster0.p7u6nwc.mongodb.net/users"  
  )  
  .then(() => {  
    console.log("Connected to MongoDB");  
  })  
  .catch((err) => {  
    console.log(err + "Failed to connect to MongoDB");  
  });  
  
app.get("/hello", (req, res) => {  
  res.send("hello");  
});  
  
app.post("/api/register", (req, res) => {  
});  
  
app.listen(1337, () => {  
  console.log("server started");  
});
```

Steps ( code to be written inside '/api/register/' )

1. Create a new User Model Object

2. Pass the Data in the User Model with the data given by React from **req.body**
3. Save the data in the database
4. Add some hashing and validations from API

#### Create a new User Model Object

```
app.post("/api/register", (req, res) => {  
  const newUser = new User({  
  });  
});
```

#### Pass the Data in the User Model with the data given by React from req.body

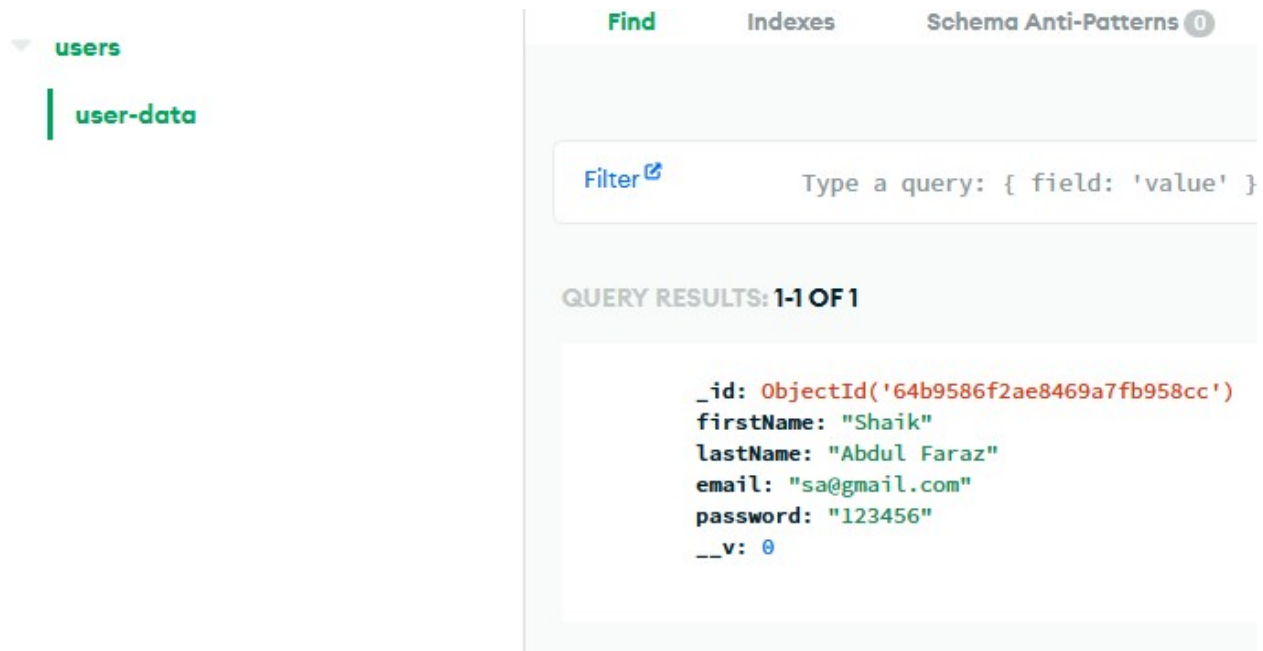
```
app.post("/api/register", (req, res) => {  
  const newUser = new User({  
    firstName: req.body.formData.firstName,  
    lastName: req.body.formData.lastName,  
    email: req.body.formData.email,  
    password: req.body.formData.password,  
  });  
});
```

#### Save the data in the database

```
app.post("/api/register", (req, res) => {  
  const newUser = new User({  
    firstName: req.body.formData.firstName,  
    lastName: req.body.formData.lastName,  
    email: req.body.formData.email,  
    password: req.body.formData.password,  
  });  
  newUser.save();  
  console.log(newUser);  
  res.json({ status: "ok" });  
});
```

Now if you click on register button from UI → in the DB it will get reflected as follows





But there is one problem with this → password is not encrypted , it must and always be encrypted! → To do this we are going to wrap our code into a HASH which is **bcrypt** package

**Npm add bcrypt**

And import it

`bcrypt.hash()` is a function used to hash the password before storing it in the database. The hashing process converts the plain text password into a fixed-length string of characters that is not reversible. This is an essential security measure to protect users' passwords from being exposed in case of a data breach

```
app.post("/api/register", (req, res) => {  
  bcrypt.hash(req.body.formData.password, 10).then((hash) => {  
  
    });  
});
```

Now copy paste our old code inside this

```
app.post("/api/register", (req, res) => {  
  bcrypt.hash(req.body.formData.password, 10).then((hash) => {  
    const newUser = new User({  
      firstName: req.body.formData.firstName,  
      lastName: req.body.formData.lastName,  
      email: req.body.formData.email,
```

```

        password: hash,
    });
    newUser.save();
    console.log(newUser);
    res.status(201).json({
        message: "User added successfully",
    });
});
});

```

Now let's register a user and test this

```

_id: ObjectId('64b95dc396b44a904abd8608')
firstName: "Shaik"
lastName: "Abdul Faraz"
email: "sal22@gmail.com"
password: "$2b$10$seG/C8DlPtuzi3QdquXIU.QQkMvV2npjCjeLRN7z57Bg9TQ/112aG"
__v: 0

```

That's amazing !

Now let's do some error handling

```

app.post("/api/register", (req, res) => {
    bcrypt
        .hash(req.body.formData.password, 10)
        .then((hash) => {
            const newUser = new User({
                firstName: req.body.formData.firstName,
                lastName: req.body.formData.lastName,
                email: req.body.formData.email,
                password: hash,
            });
            console.log(newUser);
            return newUser.save();
        })
        .then((savedUser) => {
            console.log(savedUser);
            res.status(201).json({
                message: "User added successfully",
            });
        })
        .catch((error) => {
            if (error.name === "MongoError" && error.code === 11000) {

```

```

    // Duplicate key error, email is not unique
    res.status(409).json({
      error: "Email already exists",
    });
  } else {
    // Other errors
    res.status(400).json({
      error: "Failed to add user",
    });
  }
});
});
});

```

Now let's create an END point for Login Functionality

### Login Component js

```

import React, { useState } from "react";
import './Login.css';
const LoginForm = () => {
  const [loginData, setLoginData] = useState({ email: "", password: "" });
  const [errors, setErrors] = useState({});
  const [formSubmitted, setFormSubmitted] = useState(false);

  const handleChange = (e) => {
    const { name, value } = e.target;
    setLoginData((prevData) => ({
      ...prevData,
      [name]: value,
    }));
  };

  async function handleSubmit(e) {
    e.preventDefault();

    // Basic validation
    const newErrors = {};
    if (!loginData.email) {
      newErrors.email = "Email is required.";
    }
    if (!loginData.password) {
      newErrors.password = "Password is required.";
    }

    setErrors(newErrors);

    // If no errors, proceed with login logic here (e.g., send data to the server)
    if (Object.keys(newErrors).length === 0) {
      const response = await fetch("http://localhost:1337/api/login", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
      },

```

```

        body: JSON.stringify({
            loginData,
        }),
    });
    const data = await response.json();
    setFormSubmitted(true);
    setErrors(response);
    console.log(response.status);
  }
}
return (
  <form onSubmit={handleSubmit}>
    <div>
      <label htmlFor="email">Email </label>
      <input
        type="email"
        id="email"
        name="email"
        value={loginData.email}
        onChange={handleChange}
      />
      {errors.email && <div className="error">{errors.email}</div>}
    </div>
    <div>
      <label htmlFor="password">Password</label>
      <input
        type="password"
        id="password"
        name="password"
        value={loginData.password}
        onChange={handleChange}
      />
      {formSubmitted && errors.password && <div className="error">{errors.password}</div>}
    </div>
    <button type="submit">Login</button>
    {formSubmitted && errors.status === 200 ? (
      <div className="login-banner success">
        <p>Login successful!</p>
      </div>
    ) : (
      <div className="login-banner failure">
        <p>Failed to login. Please try again.</p>
      </div>
    )}
  </form>
);
};

export default LoginForm;

```

### Login CSS

```

.login-banner {
  display: flex;
  justify-content: center;
  align-items: center;
  border-radius: 5px;
  font-size: 18px;
}

```

```

    font-weight: bold;
    color: #fff;
    margin: 10px;
  }

  .success {
    background-color: #4caf50;
  }

  .failure {
    background-color: #f44336;
  }
}

```

## API END POINT FOR LOGIN

```

app.post("/api/login", (req, res) => {
  const { email, password } = req.body.formData;

  // Find the user by email in the database
  User.findOne({ email })
    .then((user) => {
      if (!user) {
        // User with the provided email not found
        return res.status(401).json({
          error: "Authentication failed. Email or password is incorrect.",
        });
      }

      // Compare the provided password with the hashed password in the database
      bcrypt.compare(password, user.password, (err, result) => {
        if (err) {
          // Error in bcrypt comparison
          return res.status(500).json({ error: "Internal Server Error" });
        }

        if (!result) {
          // Passwords don't match
          return res.status(401).json({
            error: "Authentication failed. Email or password is incorrect.",
          });
        }

        // Passwords match, create a JWT token for authentication
        const token = jwt.sign(
          {
            userId: user._id,
            email: user.email,
          },
          "your_secret_key", // Replace this with your own secret key for JWT signing
          {
            expiresIn: "1h", // Token expiration time (optional)
          }
        );

        // Send the token in the response
        res.status(200).json({
          message: "Authentication successful",
          token: token,
        });
      });
    });
});

```

```
    });  
  })  
  .catch((error) => {  
    res.status(500).json({ error: "Internal Server Error" });  
  });  
});
```