1.llustrate the concept of component composition in React, with the process of how props used to pass data between components, how can we access the nested content using this. props. Children.

Solution :

- React classes let us create real components (as opposed to the templated HTML that we saw in the previous chapter), reuse them within other components, handle events, and much more.
- React classes are created by extending React.Component.
- There is also a non-ES2015 class way of doing this by calling React.createClass.
- It's possible to build a component that uses other user-defined components as well. This is called component composition, and it is one of the most powerful features of React.
- It lets us split the UI into smaller independent pieces so that we can reason about each piece in isolation.
- Using components rather than building the UI in a monolithic fashion also encourages reuse.
- A component takes inputs (called properties) and its output is the rendered UI of the component.

- Composing Components:

```
const contentNode = document.getElementById('contents');

class IssueFilter extends React.Component {
  render() {
    return (
      <div>This is a placeholder for the Issue Filter.</div>
    )
  }
}

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

class IssueAdd extends React.Component {
  render() {
    return (
      <div>This is a placeholder for an Issue Add entry form.</div>
    )
  }
}

class IssueList extends React.Component {
  render() {
    return (
      <div>
        <h1>Issue Tracker</h1>
        <IssueFilter />
        <hr />
        <IssueTable />
        <hr />
        <IssueAdd />
      </div>
    );
  }
}

ReactDOM.render(<IssueList />, contentNode);  // Render the component inside ←
                                                 the content Node
```

Using Properties

- Any data passed in from the parent can be accessed in the child component through a special variable, this.props.
- So, to access a property called issue_title, we use this.props.issue_title.
- For example, in the IssueRow child component, we access the property in a JSX snippet like this:

  ...
  <td>{this.props.issue_title}</td>
  ...
  And, to pass this data, use XML- or HTML-like attributes in the parent.
- For example, to pass the property issue_title, in the parent, do this:

  ...
  <IssueRow issue_title="Title of the first issue" />
  ...
  We can pass not only strings but also JavaScript objects and other data types.
- Any JavaScript expression can be passed along by using curly braces ({}) instead of quotes.
- The new IssueRow component and the modified IssueTable component is shown below

```
class IssueRow extends React.Component {
  render() {
    const borderedStyle = {border: "1px solid silver", padding: 4};
    return (
      <tr>
        <td style={borderedStyle}>{this.props.issue_id}</td>
        <td style={borderedStyle}>{this.props.issue_title}</td>
      </tr>
    )
  }
}
```

2a)Discuss how react uses JSX for rendering UI components. What are the benefits of using JSX over plain JavaScript in react applications.

The transformation of JSX to JavaScript happens at runtime is inefficient and quite unnecessary.

Transformation at the build stage in the development, will deploy a ready-to-use distribution of the application.

Separate Script File
First, we need to separate out the JSX script from the all-in-one index.html, and refer to it as an external script :App.jsx, and place it in the static directory, so that it can be referred to as /App.jsx from the browser.

The new modified files are shown below

index.html:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8" />
<title>Pro MERN Stack</title>
<script src= "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js">
</script>
<script src= "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js">
</script>
<script src= "https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.23/browser.min.js">
</script>
</head>
<body>
<div id="contents"></div><!-- this is where our component will appear -->
<script type="text/babel" src="/App.jsx"></script>
</body>
</html>
```

App.jsx: JSX Part Separated Out from the HTML

```
var contentNode = document.getElementById('contents');
var component = <h1>Hello World!</h1>;   // A simple JSX component
ReactDOM.render(component, contentNode);   // Render the component inside
                                              the content Node
```

The JSX continues to get transformed by the babel library script.

Transform

- Create a directory named src to keep all the JSX files, which will be transformed into JavaScript and place it in the static folder.
- move App.jsx into this directory.
- To transform the JSX, we need to install some babel tools.
    - babel-cli :the command line tool that invokes the transformation
    - babel-preset-react :the plugin that handles React JSX transformation

Thus execute the following command:

$ npm install --save-dev babel-cli babel-preset-react

$ node_modules/.bin/babel src --presets react --out-dir static

We also need to change index.html to replace the reference to App.jsx to App.js and indicate the new type of this script; it is now JavaScript and not JSX.

- So, let's just remove the type="text/babel" in the script specification.
- We no longer need the runtime transformer to be loaded in index.html, so we can get rid of the babel-core script library.

- These changes should be done in index.html

```
...
  <script
    src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.
    min.js"></script>
...
  <body>
    <div id="contents"></div><!-- this is where our component will appear -->
    <script src="/App.js" type="text/babel"></script>
  </body>
...
```

Automate

- npm provides an automatic way of running command-line binaries that belong to locally installed packages.
- We can define our own npm commands by specifying them in the scripts section of package.json .
- These commands can then be called using npm run <script> from the console.
- Let's add a script called "compile" whose command line is the babel command line, but without the prefix to the binary location as shown below.
- This is because npm automatically figures out the location.

$ npm run compile

- Make a few temporary changes to App.jsx, recompile it, and ensure that the changes are visible in the browser.
- To recompile, just run **npm run compile** again.

- When we work on the client-side code and change the source files frequently, we have to manually recompile it. These changes can be detected and recompiled the source into JavaScript by a option --watch.
- To make use of it, let's add another script called watch with this additional option to the babel command line.
- The final set of scripts added is shown below ,

```
...
"scripts": {
"compile": "babel src --presets react --out-dir static",
"watch": "babel src --presets react --out-dir static --watch",
"test": "echo \"Error: no test specified\" && exit 1"
},
```

Now run

npm run watch

- We will notice that it does one transform, but doesn't return to the shell.

- It's actually waiting in a permanent loop, watching for changes to the source files.
- Test out its effect by make a small change to App.jsx and save the file. We'll see that it prints out a fresh line like this for every change:
  src/App.jsx -> static/App.js

- Refresh the browser after the new line is printed when a change in App.jsx is made and ensure that the changes are reflected in the application.

```
class IssueTable extends React.Component {
  render() {
    const borderedStyle = {border: "1px solid silver", padding: 6};
    return (
      <table style={{borderCollapse: "collapse"}}>
        <thead>
          <tr>
            <th style={borderedStyle}>Id</th>
            <th style={borderedStyle}>Title</th>
          </tr>
        </thead>
        <tbody>
          <IssueRow issue_id={1}
           issue_title="Error in console when clicking Add" />
          <IssueRow issue_id={2}
           issue_title="Missing bottom border on panel" />
        </tbody>
      </table>
    )
  }
}
```

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

- We can use this wrapper to wrap any component like so:

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

- Instead of passing issue_title as a **prop**, we can **embed it as content** (children) inside the IssueRow tag. Example usage:

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

- Inside IssueRow, use **this.props.children** to access and render the content:

<td style={borderedStyle}>{this.props.issue_id}</td>

<td style={borderedStyle}>{this.props.children}</td>

- This method allows passing **formatted HTML content** instead of just plain text, offering more flexibility.

2b). Describe the steps involved in creating react components using ES6 class syntax. What are the essential life cycle methods used in such a component.

Step 1: Import React and Component

First, import React and the Component class from the React library.

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

Step 2: Create a Class Component

Create a class that extends the Component class.

```
class Welcome extends Component {

}
```

Step 3: Add Constructor

The constructor is used to initialize the state.

```
constructor(props) {
   super(props);

   this.state = {
      message: "Welcome to React"
   };
}
```

- super(props) is used to access parent class properties.
- this.state stores component data.

Step 4: Define render() Method

Every class component must contain a render() method.

```
render() {
   return (
      <h1>{this.state.message}</h1>
   );
}
```

The render() method returns JSX that is displayed on the screen.

Step 5: Export the Component

Export the component for use in other files.

export default Welcome;

Complete Example

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

class Welcome extends Component {

   constructor(props) {
      super(props);

      this.state = {
         message: "Welcome to React"
      };
   }

   render() {
      return (
         <div>
            <h1>{this.state.message}</h1>
         </div>
      );
   }
}

export default Welcome;
```

Essential Lifecycle Methods in React Class Components

Lifecycle methods are special methods automatically called during different stages of a component's life.

They are divided into three phases:

1. Mounting Phase
2. Updating Phase
3. Unmounting Phase

3.Write a react class component that displays a button and a counter. Each time the button is clicked, increase the count and display it. Use a constructor to initialize state and setstate() to update it.

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


const Counter = () => {
```

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

  // Simulate fetching initial data for the Counter component
  useEffect(() => {
    setTimeout(() => {
      setCount(0);  // Set initial value after 2 seconds
    }, 2000);
  }, []);

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
      <button onClick={() => setCount(count * 2)}>Double</button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
};
export default Counter;
function App() {
  return (
    <div className="App">
      <h1>Welcome to Counter App</h1>
      <Counter />
    </div>
  );
}
export default App;
```

4a). How are HTTP methods mapped to CRUD operations in REST, Explain the limitations of REST and design a RESTful service with proper URI and method usage

HTTP Methods as Actions

- To access and manipulate the resources, we use HTTP methods.
- While resources were nouns, the HTTP methods are verbs that operate on them. They map to CRUD (Create, Read, Update, Delete) operations on the resource. Table below shows commonly used mapping of CRUD operations to HTTP methods and resources.

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

. HEAD and OPTIONS are also valid verbs that give out information about the resources rather than actual data.

➢ **Other HTTP Methods in REST**

- **DELETE and PUT** on a **collection URI** can delete/modify the entire collection.
  - **Not commonly used** in practice.
- **HEAD and OPTIONS** are also valid HTTP methods:
  - Provide **metadata** or **information** about the resource.
  - Commonly used in **public APIs** accessed by many clients.

➢ **REST Limitations – Not Strictly Defined for:**

1. **Filtering, Sorting, Pagination**
   - Typically handled via the **query string**.
   - No standard format — **implementation-specific**.
2. **Field Selection in READ Operations**
   - REST does **not define how to select specific fields** in the response.
3. **Expanding Embedded Objects**
   - No standard way to specify which **nested/related resources** to include in a READ.
4. **Partial Updates in PATCH**
   - No strict rule on **how to specify which fields** should be updated.
5. **Data Representation Format**
   - REST does **not enforce a specific format** like JSON or XML.
   - We can choose the format for both reading and writing data.

Thus Most APIs are **REST-like**, not strictly REST.

- Due to inconsistencies:
  - **Tooling support is limited**.

- Developers need to handle many **common patterns manually**
-

4b) How does GraphQL handle over-fetching and under-fetching, Design a GraphQL schema to replace multiple REST endpoints with a single query interface.

GraphQL

- **GraphQL addresses REST shortcomings such as:**
  - **Lack of control over response structure**
  - **Multiple endpoint calls for related data**
- **It is ideal for different clients (e.g., mobile vs. desktop) that require different data formats and amounts of data.**

Core Features of GraphQL

1. Field Specification

- **Clients explicitly specify which fields they need.**
- **Prevents:**
  - **Over-fetching → receiving unnecessary data**
  - **Under-fetching → needing additional requests**
- **Improves network efficiency.**
- **No API versioning is required when new fields are added.**

Example GraphQL Query

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

Response

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

Only the requested fields are returned.

---

2. Graph-Based Structure

- **GraphQL models relationships naturally.**
- **Supports nested queries.**
- **Related data can be fetched in a single request.**

Example Query

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

```
  }
}
```

Response

```
{
 "data": {
  "user": {
   "name": "Alice",
   "issues": [
    {
     "title": "Login Bug",
     "status": "Open"
    },
    {
     "title": "UI Fix",
     "status": "Closed"
    }
   ]
  }
 }
}
```

1. **Single Endpoint**
   - **REST APIs** require multiple endpoints (/users, /orders, /products).
   - **GraphQL APIs** use a **single endpoint** (e.g., /graphql).
   - Replaces multiple REST endpoints.
   - Simplifies client-server communication.

2. **Strongly Typed**
   - All types, fields, and arguments have **strict types**.
   - Prevents errors by **validating queries**.
   - Enforces **required and optional fields**.
   - Schema is written using **GraphQL Schema Language**.
3. **Introspection**
   - GraphQL server can describe its **schema and types**.
   - Enables tools like:
     - **GraphQL Playground / Apollo Studio**
     - **Code generators** for typed languages
     - **Query explorers** for learning and testing
4. **Libraries & Tools**
   - **graphql-js**: Core GraphQL library for JavaScript.
   - **express-graphql**: Connects GraphQL to Express apps.
   - **graphql-tools** & **apollo-server**: Advanced tools for modular schemas, custom scalars, etc.
   - Used for building robust and scalable APIs in modern web applications.

Thus, GraphQL provides:

- **Granular control** over data.
- **Single endpoint** with rich querying capabilities.
- **Type safety**, **introspection**, and **tooling support**.
- A better fit for modern apps with **dynamic UI requirements**.

- The new contents of server.js are shown below,

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

let aboutMessage = "Issue Tracker API v1.0";
const typeDefs = `
  type Query {
    about: String!
  }
  type Mutation {
    setAboutMessage(message: String!): String
  }
`;

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

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


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

const app = express();

app.use(express.static('public'));

server.applyMiddleware({ app, path: '/graphql' });

app.listen(3000, function () {
  console.log('App started on port 3000');
});
```

5a. Write Mongo shell commands to perform the following operations:

Insert three employee documents with different fields. Update one document to add a middle name. Delete one document by-id

Create an index on the age field

Query employees whose age is greater than 30.

1. Insert Three Employee Documents

```
db.employee.insertMany([
  {
    _id: 1,
    firstName: "Rahul",
    lastName: "Sharma",
    age: 25,
    department: "HR"
  },

  {
    _id: 2,
    firstName: "Anita",
    lastName: "Verma",
    age: 35,
    department: "Finance"
  },

  {
    _id: 3,
    firstName: "Kiran",
    lastName: "Patil",
    age: 40,
    department: "IT"
  }
]);
```

2. Update One Document to Add Middle Name

```
db.employee.updateOne(
  { _id: 2 },
  {
    $set: {
      middleName: "Kumar"
    }
  }
);
```

3. Delete One Document by ID

```
db.employee.deleteOne(
  { _id: 1 }
);
```

4. Create an Index on Age Field

```
db.employee.createIndex(
  { age: 1 }
);
```

- 1 indicates ascending order index.

5. Query Employees Whose Age is Greater Than 30

```
db.employee.find(
  { age: { $gt: 30 } }
);
```

- $gt means “greater than”.

5b. What problem does Hot Module Replacement solve, how is HMR implemented using webpack-hot-middleware

Hot Module Replacement (HMR) is a feature in Webpack that allows modules to be updated in a running application without refreshing the entire page. It improves the development experience by updating only the changed modules while preserving the application state.

Problem Solved by Hot Module Replacement (HMR)

Before HMR, whenever developers made changes to source code, the browser had to reload the entire page. This caused several problems:

1. Loss of Application State

Form inputs, application data, and UI state were lost after every refresh.

2. Slow Development Process

Full page reloads increased waiting time during development.

3. Reduced Productivity

Developers had to repeatedly navigate back to the same page after every change.

4. Poor Debugging Experience

Continuous refreshes interrupted testing and debugging.

HMR solves these problems by updating only the modified module without reloading the complete application.

6a) Illustrate the difference between updateOne(), updateMany(), replaceOne(), and updateMany() operation with examples.

1. updateOne()

Definition

updateOne() updates only the first document that matches the given condition.

Syntax

```
db.collection.updateOne(
  { condition },
  { $set: { field: value } }
);
```

2. updateMany()

Definition

updateMany() updates all documents that match the specified condition.

Syntax

```
db.collection.updateMany(
  { condition },
  { $set: { field: value } }
);
```

3. replaceOne()

Definition

replaceOne() replaces an entire document with a new document except the _id field.

Syntax

```
db.collection.replaceOne(
  { condition },
  { newDocument }
);
```

6b) How does Webpack's DefinePlugin help in injecting environment variables, mention the limitations of using DefinePlugin, and what optimizations does Webpack apply in production mode

DefinePlugin is a built-in Webpack plugin used to create global constants during the compilation process. It is mainly used to inject environment variables such as development or production settings into the application.

It replaces variables at build time, allowing developers to configure applications for different environments**.**

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

plugins: [
  new webpack.DefinePlugin({
    'process.env.NODE_ENV': JSON.stringify('production')
  })
]
```

Example of Injecting Environment Variables

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

module.exports = {
mode: 'development',

plugins: [
new webpack.DefinePlugin({
'process.env.API_URL': JSON.stringify('https://api.example.com'),
```

```
'process.env.NODE_ENV': JSON.stringify('development')
})
]
};
```

Using Variables in Application

```
console.log(process.env.API_URL);

if (process.env.NODE_ENV === 'development') {
  console.log("Development Mode");
}
```