FULL STACK WEB DEVELOPMENT

REACT:

```
function MyButton() {
  return (
    <button>I'm a button</button>);}
export default function MyApp() {
  return (
    <div>
      <h1>Welcome to my app</h1><MyButton /></div>);}

import { createRoot } from 'react-dom/client';
// Clear the existing HTML content
document.body.innerHTML = '<div id="app"></div>';
// Render your React component instead
const root = createRoot(document.getElementById('app'));
root.render(<h1>Hello, world</h1>);

const user = {name: 'Hedy Lamarr',
  imageUrl: 'https://i.imgur.com/yXOvdOSs.jpg',imageSize: 90,};
export default function Profile() {
  return (
    <><h1>{user.name}</h1>
      <img className="avatar" src={user.imageUrl} alt={'Photo of ' + user.name}
        style={{width: user.imageSize, height: user.imageSize}}/></>);}
```

Rendering Lists:

```
const products = [
  { title: 'Cabbage', isFruit: false, id: 1 },
  { title: 'Garlic', isFruit: false, id: 2 },
  { title: 'Apple', isFruit: true, id: 3 },];
export default function ShoppingList() {
  const listItems = products.map(product =>
    <li key={product.id} style={{color: product.isFruit ? 'magenta' : 'darkgreen'}}>
{product.title}
    </li>);
  return ( <ul>{listItems}</ul> );}
```

Updating the screen:

```
import { useState } from 'react';
export default function MyApp() {
  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>);}
function MyButton() {
  const [count, setCount] = useState(0);
  function handleClick() {
    setCount(count + 1);}
  return (<button onClick={handleClick}>
      Clicked {count} times</button>);}
```

React Components:

The createClass() factory method receives an object which defines methods for the React component. Whereas the getInitialState() function is used to set an initial state for the React component, the mandatory render() method

is there to display the output with JSX. Additional "methods" (e.g. onChange()) are added by passing more functions to the object.
Lifecycle methods for side-effects are available as well. For instance, in order to write every time the value from the input field to the browser's local storage, we could make use of the componentDidUpdate() lifecycle method by passing a function to the object with an object key named after a React lifecycle method. In addition, the value can be read from the local storage when the component receives its initial state:

```
var App = React.createClass({
 getInitialState: function() {
  return {
   value: localStorage.getItem('myValueInLocalStorage') || '',
  };
 },

 componentDidUpdate: function() {
  localStorage.setItem('myValueInLocalStorage', this.state.value);
 },

 onChange: function(event) {
  this.setState({ value: event.target.value });
 },

 render: function() {
  return (
   <div>
    <h1>Hello React "createClass" Component!</h1>

    <input
     value={this.state.value}
     type="text"
     onChange={this.onChange}
    />

    <p>{this.state.value}</p>
   </div>
  );
 },});
```

React Class Components were introduced with JavaScript ES6, because JS classes were made available to the language. Sometimes they are called **React ES6 class components** as well.

```
class App extends React.Component {
 constructor(props) {
  super(props);

  this.state = {
   value: '',
  };

  this.onChange = this.onChange.bind(this);
 }

 onChange(event) {
  this.setState({ value: event.target.value });
 }

 render() {
  return (
   <div>
    <h1>Hello React ES6 Class Component!</h1>

    <input
     value={this.state.value}
     type="text"
     onChange={this.onChange}
    />

    <p>{this.state.value}</p>
   </div>
  );
 }
}
```

A React Component written with a JavaScript class comes with methods like the class constructor -- which is primarily used in React to set initial state or to bind methods -- and the mandatory render method to return JSX as output. All the internal React Component logic comes from the extends React.Component via object-oriented inheritance that is used in the class component.

React Function Components are the equivalent of React Class Components but expressed as functions instead of classes. In the past, it wasn't possible to use state or side-effects in Function Components -- that's why they were called **Functional Stateless Components** -- but that's not the case anymore with React Hooks which rebranded them to Function Components.

React Hooks bring state and side-effects to React Function Components. React comes with a variety of built-in hooks, but also the ability to create custom hooks for yourself or others. Let's see how the previous React Class Component can be used as a React Function Component:

```
const App = () => {
 const [value, setValue] = React.useState('');

 const onChange = event => setValue(event.target.value);

 return (
  <div>
   <h1>Hello React Function Component!</h1>

   <input value={value} type="text" onChange={onChange} />

   <p>{value}</p>
  </div>
 );
};
```

The previous code only shows the Function Component with the input field. Since component state is needed to capture the state of the input field's value, we are using the built-in React useState Hook.

****Templating with JSX:****

JSX (JavaScript XML) is a syntax extension for JavaScript, often used with React to describe what the UI should look like. JSX allows you to write HTML-like code in your JavaScript files.

~~~
import React from 'react';
const MyComponent = () => {
  return (
    <div><h1>Hello, JSX!</h1>
      <p>This is a paragraph in JSX.</p></div>);};
export default MyComponent;
~~~

Babel:

Babel is a JavaScript compiler that allows you to use the latest ECMAScript features in your code. It's often used with JSX and React to transpile modern JavaScript code into a version that can run in all browsers.

~~~
// Babel will transform this code
const arrowFunction = () => {
  console.log('Hello, Babel!');};
~~~

ES6 Literals and Expressions in JSX:

~~~
import React from 'react';
const name = 'John Doe';
const MyComponent = () => {
  return <p>Hello, {name}!</p>;};
export default MyComponent;
~~~

JSX Attributes:

~~~
import React from 'react';
const MyComponent = () => {
  const imgSrc = 'path/to/image.jpg';
  return <img src={imgSrc} alt="An example image" />;};
export default MyComponent;
~~~

Styling React Elements:

Example:

~~~
```jsx
import React from 'react';
const MyComponent = () => {
  const textStyle = {
    color: 'blue', fontSize: '16px', fontWeight: 'bold',};
  return <p style={textStyle}>Styled Text</p>;};
export default MyComponent;

```jsx
import styled from 'styled-components';
const StyledParagraph = styled.p`
  color: blue; font-size: 16px;font-weight: bold;`;
const MyComponent = () => {
  return <StyledParagraph>Styled Text</StyledParagraph>;};
export default MyComponent;
~~~

Express Integration (Example using Express and JSX):

~~~
```jsx
// Assuming you have an Express app set up
const express = require('express');
const app = express();
const port = 3000;
app.get('/', (req, res) => {
  const message = 'Hello from Express and JSX!';
  res.send(
    <div><h1>{message}</h1>
~~~

```
  </div>);});
app.listen(port, () => {
 console.log(`Server is running at http://localhost:${port}`);});
```

****States and Props:****
Props:
Props (short for properties) are used to pass data from a parent component to a child component in React. Props are immutable and are passed down the component tree.

```
// ParentComponent.js
import React from 'react';
import ChildComponent from './ChildComponent';
const ParentComponent = () => {
 const data = 'Hello from parent!';
 return <ChildComponent message={data} />;};
// ChildComponent.js
import React from 'react';
const ChildComponent = (props) => {
 return <p>{props.message}</p>;};
export default ChildComponent;
```

States:
States are used to manage mutable data within a component. Unlike props, states can be changed by the component itself using the `setState` method.

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

****Life Cycle of Components:****
In React, components have several lifecycle methods that allow you to perform actions at different stages of a component's existence. However, with the introduction of React Hooks, class component lifecycle methods are less commonly used in modern React applications.

```
import React, { Component } from 'react';
class LifecycleExample extends Component {
 constructor(props) {
  super(props);
  console.log('Constructor called');}
 componentDidMount() {
  console.log('Component did mount');}
 componentDidUpdate() {
  console.log('Component did update');}
 componentWillUnmount() {
  console.log('Component will unmount');}
 render() {
  console.log('Render called');
  return <p>Component Lifecycle Example</p>;}}
export default LifecycleExample;
```

****Stateless (Functional) Components:****
Stateless components, also known as functional components, are simpler components that do not manage state. They receive data through props and return React elements.

```
// StatelessComponent.js
import React from 'react';
const StatelessComponent = ({ message }) => {
  return <p>{message}</p>;};
export default StatelessComponent;
```
Functional components are often used with React Hooks to introduce state and other features without using class components.
```
import React, { useState } from 'react';
const FunctionalComponent = () => {
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount(count + 1);};
  return (
    <div><p>Count: {count}</p>
      <button onClick={increment}>Increment</button></div>);};
export default FunctionalComponent;
```
It's worth noting that with the introduction of React Hooks in functional components, there's less need for class components and lifecycle methods in modern React development. Hooks provide a way to use state and other React features in functional components.

****Event Handling:****
```
import React, { useState } from 'react';
const EventHandlingExample = () => {
  const [message, setMessage] = useState('Click the button');
  const handleClick = () => {
    setMessage('Button clicked!');};
  return (
    <div><p>{message}</p>
      <button onClick={handleClick}>Click me</button></div>);};
export default EventHandlingExample;
```
****Error Handling:****
```
import React, { Component } from 'react';
class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };}

  static getDerivedStateFromError(error) {
    return { hasError: true };}
  componentDidCatch(error, errorInfo) {
    logErrorToMyService(error, errorInfo);}
  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }return this.props.children;
  }}<ErrorBoundary><MyComponent /></ErrorBoundary>;
```
****Routers:****
```
import React from 'react';
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';
const Home = () => <h2>Home</h2>;
const About = () => <h2>About</h2>;
const App = () => (
  <Router><div>
      <nav><ul>
```

```
        <li><Link to="/">Home</Link></li>
      <li><Link to="/about">About</Link>
      </li></ul></nav><hr />
    <Route exact path="/" component={Home} />
    <Route path="/about" component={About} />
  </div></Router>);
export default App;
```
****Single Page Application (SPA):****

A Single Page Application (SPA) is a web application or website that interacts with the user by dynamically rewriting the current page, rather than loading entire new pages from the server. React is often used to build SPAs.

```
// App.js
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Home from './Home';
import About from './About';
import NotFound from './NotFound';
const App = () => (
  <Router><Switch>
    <Route exact path="/" component={Home} />
    <Route path="/about" component={About} />
    <Route component={NotFound} />
  </Switch></Router>);
export default App;
```
NODE.JS:

Remember to install the necessary dependencies, like `react-router-dom`, using:

```
npm install react-router-dom
```
****Node.js Setup:****

1. **Download and Install:**
   Visit the official Node.js website (https://nodejs.org/) and download the installer for your operating system. Run the installer to install Node.js and npm (Node Package Manager).
2. **Verify Installation:**
   Open a terminal or command prompt and run the following commands to verify the installation.

****Node Life Cycle:****

Node.js follows a life cycle that includes initializing, executing the script, and terminating.

- **Initialization:**
  Node.js initializes its environment, including loading modules and preparing to execute the script.
- **Execution:**
  The script is executed, and asynchronous events are handled using an event-driven architecture.
- **Termination:**
  Node.js terminates when there are no more callbacks to perform.

****REPL (Read-Eval-Print Loop):****

Node.js comes with a built-in REPL environment that allows you to interact with Node.js directly in the terminal. To start the REPL, open a terminal and type `node`. You can then enter JavaScript commands, and they will be executed immediately.

****Node Modules - FS, HTTP, URL:****

Node.js provides several built-in modules. Here are examples of using `fs` (File System), `http`, and `url` modules:

```
// Example using fs module to read a file
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;} console.log(data);});
// Example using http module to create a simple server
```

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!');});
server.listen(3000, () => {
  console.log('Server listening on port 3000');
});
// Example using url module to parse a URL
const url = require('url');
const urlString = 'https://www.example.com/path?query=value';
const parsedUrl = url.parse(urlString, true);
console.log(parsedUrl);
```
****NPM (Node Package Manager):****
NPM is the package manager for Node.js. You can use it to install and manage third-party packages. To initialize a new project and create a `package.json` file:
```
npm init
```
To install a package, use:
```
npm install package-name
```
****Redirecting Requests, Callbacks, Events:****
Example using the `http` module to create a simple server that redirects requests:
```
const http = require('http');
const url = require('url');
const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url, true);
  if (parsedUrl.pathname === '/redirect') {
    res.writeHead(301, { Location: 'https://www.example.com' });
    res.end();
  } else {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, World!');
  }});
server.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```
****Blocking and Non-Blocking Code:****
Node.js is designed to be non-blocking, allowing it to handle many connections simultaneously. Here's an example illustrating blocking vs. non-blocking code:
```
// Blocking Code
const fs = require('fs');
const data = fs.readFileSync('example.txt', 'utf8');
console.log(data);

// Non-Blocking Code
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;}
  console.log(data);
});
```
****Node.js with Express:****
Express is a popular web application framework for Node.js. To use Express, first, install it using:
```
npm install express
```

Example of a simple Express server:
```
const express = require('express');
const app = express();
const port = 3000;
app.get('/', (req, res) => {
  res.send('Hello, Express!');
});
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```
Express - Handling Requests and Responses:
Express simplifies handling requests and responses. Here's an example with route parameters and sending JSON:
```
const express = require('express');
const app = express();
const port = 3000;
app.get('/', (req, res) => {
  res.send('Hello, Express!');});
app.get('/user/:id', (req, res) => {
  const userId = req.params.id;
  res.json({ userId });});
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);});
```
These examples cover a range of Node.js concepts, from setting up the environment to using built-in modules, NPM, handling requests with Express, and understanding the event-driven, non-blocking nature of Node.js.

MONGODB:

SQL and NoSQL Concepts:

SQL (Structured Query Language):
- SQL is a relational database management system (RDBMS).
- It uses a structured schema with tables to organize and store data.
- Tables have predefined columns and each row represents a record.

NoSQL (Not Only SQL):
- NoSQL databases are non-relational and do not require a fixed schema.
- They can store and process a variety of data types (e.g., documents, key-value pairs, graphs).
- NoSQL databases are horizontally scalable, making them suitable for large amounts of distributed data.

Create and Manage MongoDB:

Install MongoDB:
1. Visit the MongoDB website (https://www.mongodb.com/try/download/community) and download the community edition.
2. Follow the installation instructions for your operating system.

Start MongoDB:
- After installation, start the MongoDB server by running the `mongod` command.

CRUD Operations on MongoDB:
MongoDB uses a document-oriented model with BSON (Binary JSON) documents. Here are basic CRUD operations:

Create (Insert):
```
db.users.insertOne({ name: 'John Doe', age: 25, email: 'john@example.com' });
```
Read (Query):
```
db.users.find({ name: 'John Doe' });
```
Update:
```
db.users.updateOne({ name: 'John Doe' }, { $set: { age: 26 } });
```
Delete:
```
db.users.deleteOne({ name: 'John Doe' });
```
MongoDB with Node.js:

Install MongoDB Node.js Driver:
npm install mongodb
Example of Connecting to MongoDB and Performing CRUD Operations:
```javascript
const MongoClient = require('mongodb').MongoClient;
const uri = 'mongodb://localhost:27017/mydatabase';
MongoClient.connect(uri, { useNewUrlParser: true, useUnifiedTopology: true }, (err, client) => {
 if (err) {
  console.error('Error connecting to MongoDB:', err);
  return;}
 const db = client.db('mydatabase');
 // Insert
 db.collection('users').insertOne({ name: 'John Doe', age: 25, email: 'john@example.com' });
 // Find
 db.collection('users').findOne({ name: 'John Doe' }, (err, result) => {
  console.log('User found:', result);});
 // Update
 db.collection('users').updateOne({ name: 'John Doe' }, { $set: { age: 26 } });
 // Delete
 db.collection('users').deleteOne({ name: 'John Doe' });
 // Close the connection
 client.close();});
```
Services Offered by MongoDB:
MongoDB Atlas:
- MongoDB's cloud database service.
- Offers automated backups, scaling, and security features.
- Provides a free tier for small applications.
MongoDB Compass:
- GUI for MongoDB.
- Allows visualizing and manipulating data.
- Offers performance optimization tools.
MongoDB Stitch:
- Serverless platform for MongoDB.
- Provides backend services like authentication, functions, and triggers.
- Enables building full-stack applications without managing servers.
These services enhance MongoDB's functionality and make it easier to manage databases, whether on-premises or in the cloud. Integrating MongoDB with Node.js allows for seamless development of applications that use MongoDB as the data store.
WEB SERVICES:
Web Services Architecture:

Web services provide a standard way of interoperability between different software applications running on various platforms. The architecture typically consists of:
- **Service Provider:** Publishes the web service.
- **Service Consumer:** Consumes or uses the web service.
- **UDDI (Universal Description, Discovery, and Integration):** A directory service for registering and locating web services.
- **WSDL (Web Services Description Language):** Describes the web service interface.
Web Services Technologies - SOAP, REST:
SOAP (Simple Object Access Protocol):
- A protocol for exchanging structured information in web services.
- Communication is XML-based and follows a strict specification.
- Uses HTTP, SMTP, or other protocols for message transmission.

REST (Representational State Transfer):
- An architectural style for designing networked applications.
- Utilizes standard HTTP methods (GET, POST, PUT, DELETE).
- Emphasizes stateless communication and resource-oriented architecture.

Node.js REST API:

Install Necessary Packages:

```
npm init -y
npm install express body-parser
```

Example of Building a Simple REST API with Node.js:

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const port = 3000;
app.use(bodyParser.json());
const items = [];
// Get all items
app.get('/items', (req, res) => {
  res.json(items);});
// Add a new item
app.post('/items', (req, res) => {
  const newItem = req.body;
  items.push(newItem);
  res.status(201).json(newItem);});
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```

JSON Web Tokens (JWT) for Authentication:

JWT is a compact, URL-safe means of representing claims to be transferred between two parties. In the context of authentication, JWTs are often used to securely transmit information between parties.

Install Necessary Package:

```
npm install jsonwebtoken
```

Example of Using JWT for Authentication in a Node.js Application:

```
const express = require('express');
const jwt = require('jsonwebtoken');
const bodyParser = require('body-parser');
const app = express();
const port = 3000;
app.use(bodyParser.json());
const secretKey = 'your-secret-key';
// Middleware to verify JWT
const authenticateToken = (req, res, next) => {
  const token = req.headers['authorization'];
  if (!token) {
    return res.sendStatus(401);
  }
  jwt.verify(token, secretKey, (err, user) => {
    if (err) {
      return res.sendStatus(403);}
    req.user = user;
    next();});};
// Generate and send JWT on login
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  // Validate username and password (for demonstration purposes, hardcoding)
  if (username === 'user' && password === 'password') {
    const token = jwt.sign({ username }, secretKey);
    res.json({ token });
  } else {
    res.sendStatus(401);}});
// Protected route using JWT
app.get('/protected', authenticateToken, (req, res) => {
  res.json({ message: 'This is a protected route', user: req.user });});
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```
****Microservices Architecture:****
Microservices architecture involves breaking down a monolithic application into smaller, independent services. Each service performs a specific business function and communicates with others through well-defined APIs. Some key characteristics include:
- **Independence:** Each microservice operates independently and has its own database.
- **Decentralized Data Management:** Each service manages its data independently.
- **Scalability:** Microservices can be individually scaled based on demand.
- **Resilience:** Failure of one microservice does not affect the entire application.
Microservices can be implemented using various technologies, and communication between them often occurs through APIs (RESTful or other protocols).
SERVERLESS COMPUTING:
****Serverless Computing:****
Serverless computing is a cloud computing model that allows you to build and run applications without the need to manage infrastructure. In a serverless architecture, the cloud provider automatically handles scaling, maintenance, and availability, allowing developers to focus on writing code.
****AWS Services for Serverless Computing:****
AWS Lambda:
- AWS Lambda is a serverless compute service that runs code in response to events.
- It supports multiple programming languages, including Node.js, Python, Java, and more.
- You pay only for the compute time consumed by your code.
API Gateway:
- Amazon API Gateway is a fully managed service for building, deploying, and scaling APIs.
- It can be used to create RESTful APIs that integrate with AWS Lambda functions.
AWS S3:
- Amazon S3 (Simple Storage Service) is an object storage service that offers industry-leading scalability, data availability, security, and performance.
- It can be used to store and retrieve data in serverless applications.
AWS DynamoDB:
- Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance.
- It is often used to store and retrieve data in serverless applications.
****AWS Lambda Example:****
Let's create a simple AWS Lambda function using the AWS Management Console.
1. **Go to AWS Lambda Console:**
   - Open the AWS Lambda console (https://console.aws.amazon.com/lambda/).
2. **Create a Function:**
   - Click on the "Create function" button.
   - Choose "Author from scratch."
   - Enter a name for your function (e.g., "HelloWorld").
   - Choose Node.js 14.x as the runtime.
   - Click on "Create function."
3. **Add Code:**

- Scroll down to the "Function code" section.
- Replace the default code with the following Node.js code:

```
exports.handler = async (event) => {
  const response = {
    statusCode: 200,
    body: JSON.stringify('Hello from Lambda!'),};
  return response; };
```

4. **Test the Function:**
   - Click on the "Test" button.
   - Enter a name for the test event and click on "Create."
   - Click on "Test" again to run the test.

****Use-Cases for Serverless Applications:****

1. Web Application Backends:
- Use AWS Lambda to handle backend functions (e.g., user authentication, file uploads).
- Combine with API Gateway for creating RESTful APIs.

2. Real-time Data Processing:
- Process streaming data from sources like Amazon Kinesis.
- Perform real-time analytics and processing.

3. Scheduled Jobs:
- Schedule and automate tasks using AWS CloudWatch Events.
- Examples include data backups, report generation.

4. Image and Video Processing:
- Use Lambda to process and resize images uploaded to S3.
- Generate thumbnails or apply filters.

****Web Application Deployment in AWS:****

Deploying a web application in AWS typically involves using services like Amazon S3 for static assets, AWS Lambda for serverless functions, and API Gateway for handling HTTP requests.

Example Deployment Steps:

1. **Static Assets:**
   - Upload your HTML, CSS, and client-side JavaScript to an S3 bucket.
2. **Serverless Backend:**
   - Create AWS Lambda functions for server-side logic.
   - Set up an API Gateway to expose HTTP endpoints.
3. **Database Integration:**
   - Use a database service like DynamoDB for data storage.
4. **Domain and SSL:**
   - Register a domain using Amazon Route 53.
   - Set up SSL with AWS Certificate Manager.
5. **Content Delivery:**
   - Use Amazon CloudFront for content delivery and caching.
6. **Monitoring and Logging:**
   - Set up AWS CloudWatch for monitoring and logging.
7. **CI/CD (Optional):**
   - Implement continuous integration and deployment with AWS CodePipeline.

Deploying a web application in the cloud provides scalability, reliability, and cost-effectiveness. AWS provides a variety of services to support different aspects of web application development and deployment.

This overview and example provide a starting point for understanding Serverless Computing with AWS and deploying web applications in the cloud. You can explore more advanced features and configurations based on your specific application requirements.