IMPORTANT QUESTIONS

Q1.Build Search filter in React

React code to build a simple search filter functionality to display a filtered list based on the search query entered by the user.

The following are the steps to create a Search filter using React JS:

1. Declare React states for search input values.
2. Create HTML input text for entering search term and update state in onChange function.
3. Add Array.filter() on list of items with search term value.

```
// App.js
import React from 'react';
import SearchFilter from './SearchFilter';
const App = () => {
  return (
    <div>
      <h1>Search Filter Example</h1>
      <SearchFilter />
    </div>);};
export default App;

// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

//SearchFilter.js
import React, { useState } from 'react';
const SearchFilter = () => {
  const [searchTerm, setSearchTerm] = useState('');
  const [data, setData] = useState([
    'Apple',
    'Banana',
    'Cherry',
    'Date',
    'Grapes',
    'Lemon',
    'Orange',
    'Peach',
  ]);
  const handleSearch = (event) => {
    setSearchTerm(event.target.value);
  };
  const filteredData = data.filter((item) =>
    item.toLowerCase().includes(searchTerm.toLowerCase()));
  return (
    <div>
      <h1>Search Filter in React</h1>
```

```
      <input
        type="text"
        placeholder="Search..."
        value={searchTerm}
        onChange={handleSearch}/>
      <ul>
        {filteredData.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul></div>);};
export default SearchFilter;
```

Q2. Simple Login form in React

React code for simple login form where the user login by entering their username and password. The form inputs are validated to check if correct information is entered and the error messages are the validation fails. The login form is hidden and the “Welcome, ${name}” message is shown when the user login is successful.

The following are the steps to create a simple login form using React JS:

1. Create name, email and password input form elements.
2. React States to store user input values.
3. Add form validation for compare name, email and password with correct values.
4. Display “Welcome, ${name}” if login is successful, else display the error message.

```
// App.js
import React from 'react';
import SearchFilter from './LoginForm';
const App = () => {
  return (
    <div>
      <h1>Search Filter Example</h1>
      <SearchFilter />
    </div>);};
export default App;

//LoginForm.js
import React, { useState } from 'react';
const LoginForm = () => {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [loggedIn, setLoggedIn] = useState(false);
  const [error, setError] = useState('');
  const handleLogin = () => {
    const correctName = 'Sivani';
    const correctEmail = 'ksisivani04@gmail.com';
    const correctPassword = 'password123';
    if (name === correctName && email === correctEmail && password === correctPassword) {
      setLoggedIn(true);
      setError('');
    } else {
      setLoggedIn(false);
      setError('Invalid credentials. Please try again.');}};
  return (
    <div>
      <label>Name:
        <input type="text" value={name} onChange={(e) => setName(e.target.value)} /></label>
```

```
      <br />
      <label>Email:
        <input type="text" value={email} onChange={(e) => setEmail(e.target.value)} /></label>
      <br />
      <label>Password:
        <input type="password" value={password} onChange={(e) => setPassword(e.target.value)} />
      </label>
      <br />
      <button onClick={handleLogin}>Login</button>
      {loggedIn ? (
        <p>Welcome, {name}!</p>
      ) : (
        <p style={{ color: 'red' }}>{error}</p>
      )}
    </div>);};
export default LoginForm;

// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root'));
```

Q3. Simple counter exercise

Creating a simple counter using React which increments or decrements count dynamically on-screen as the user clicks on the button. This exercise requires knowledge of fundamental React concepts such as State, Component, etc.

We can complete the simple counter exercise with the following steps:

Create React state to store the count value.

Declare JS functions to incement or decrement the value through setState().

Add HTML buttons with onClick to JSX code.

```
// App.js

import React from 'react';
import SimpleCounter from './SimpleCounter';
const App = () => {
  return (
    <div>
      <h1>React Simple Counter Exercise</h1>
      <SimpleCounter />
    </div>
  );};
export default App;

//SimpleCounter.js

import React, { useState } from 'react';
const SimpleCounter = () => {
  const [count, setCount] = useState(0);
  const increment = () => {
```

```
    setCount(count + 1);
  };
  const decrement = () => {
    setCount(count - 1);
  };
  return (
    <div>
      <h2>Simple Counter</h2>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );};
export default SimpleCounter;

// index.js

import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Q4: react exercise to create an image slide where users can view multiple images with next/previous buttons. Additionally there is also an option to select an image from any index of the list through a click on option circle

```
// ImageSlider.js
import React, { useState } from 'react';
import './ImageSlider.css';

const ImageSlider = ({ images }) => {
 const [currentIndex, setCurrentIndex] = useState(0);

 const nextImage = () => {
  setCurrentIndex((prevIndex) => (prevIndex + 1) % images.length);
 };

 const prevImage = () => {
  setCurrentIndex((prevIndex) => (prevIndex - 1 + images.length) % images.length);
 };

 const selectImage = (index) => {
  setCurrentIndex(index);
 };

 return (
  <div className="image-slider">
   <div className="slide-container">
    <img src={images[currentIndex]} alt={`Slide ${currentIndex}`} />
    <div className="controls">
```

```
          <button onClick={prevImage}>Previous</button>
          <button onClick={nextImage}>Next</button>
        </div>
      </div>
      <div className="option-circles">
       {images.map((image, index) => (
         <div
          key={index}
          className={`option-circle ${index === currentIndex ? 'active' : ''}`}
          onClick={() => selectImage(index)}
         />
       ))}
      </div>
    </div>
  );
};

export default ImageSlider;
/* ImageSlider.css */
.image-slider {
  width: 80%;
  margin: 20px auto;
}

.slide-container {
  position: relative;
}

.controls {
  position: absolute;
  top: 50%;
  transform: translateY(-50%);
  width: 100%;
  display: flex;
  justify-content: space-between;
}

.option-circles {
  display: flex;
  justify-content: center;
  margin-top: 10px;
}

.option-circle {
  width: 10px;
  height: 10px;
  border-radius: 50%;
  background-color: #ccc;
  margin: 0 5px;
```

```
  cursor: pointer;
}

.option-circle.active {
  background-color: #333;
}
// App.js

import React from 'react';
import ImageSlider from './ImageSlider';

const images = [
  'image1.jpg',
  'image2.jpg',
  'image3.jpg',
  // Add more image URLs as needed
];

function App() {
  return (
    <div className="App">
      <ImageSlider images={images} />
    </div>
  );
}

export default App;
```

Q5:create nodeJs based script file that reads the names of the 2 files from the user (use html form to read filenames).Read the content of the first file and write into the second file.if second file is available it should append the content.if not it should create new file and add the content to it.

```
const express = require('express');
const fs = require('fs');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;

app.use(bodyParser.urlencoded({ extended: true }));

app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

app.post('/processFiles', (req, res) => {
  const { firstFileName, secondFileName } = req.body;

  // Read the content of the first file
  fs.readFile(firstFileName, 'utf8', (err, data) => {
    if (err) {
```

```
      return res.status(500).send('Error reading the first file.');
    }

    // Append or create the second file
    fs.appendFile(secondFileName, data, (err) => {
      if (err) {
        return res.status(500).send('Error writing to the second file.');
      }

      return res.send('Content successfully copied to the second file.');
    });
  });
});

app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>File Copy App</title>
</head>
<body>
  <form action="/processFiles" method="post">
    <label for="firstFileName">Enter the name of the first file:</label>
    <input type="text" id="firstFileName" name="firstFileName" required><br>

    <label for="secondFileName">Enter the name of the second file:</label>
    <input type="text" id="secondFileName" name="secondFileName" required><br>

    <button type="submit">Copy Content</button>
  </form>
</body>
</html>
```

Q6: develop a node js application to create a login page that is shown to the user as a homepage to read the username and password from the user and authenticate the user based on the data available in DB. if he is a valid user then present him the webpage with the message "Hello User username" else display an error message along with login page

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;

// Sample in-memory database for demonstration purposes
const users = [
  { username: 'user1', password: 'password1' },
```

```
  { username: 'user2', password: 'password2' },
];

app.use(bodyParser.urlencoded({ extended: true }));

// Serve the login page
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/login.html');
});

// Handle login form submission
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  // Authenticate user against the in-memory database
  const user = users.find((u) => u.username === username && u.password === password);

  if (user) {
    // Display the welcome page
    res.send(`Hello User ${username}`);
  } else {
    // Display an error message along with the login page
    res.status(401).sendFile(__dirname + '/login.html');
  }
});

app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Login Page</title>
</head>
<body>
  <h2>Login</h2>
  <form action="/login" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" required><br>

    <label for="password">Password:</label>
    <input type="password" id="password" name="password" required><br>

    <button type="submit">Login</button>
  </form>
</body>
</html>
```

Q7: describe the properties of Amazon web services as cloud computing services that make it a game changer for web applications that face uncertain but possibly explosive growth when compared to owning your services

Amazon Web Services (AWS) is a widely used cloud computing platform that provides a variety of services to help businesses build and scale web applications. AWS has several properties that make it a game-changer for web applications facing uncertain but potentially explosive growth when compared to owning and managing your own services:

1. **Scalability and Elasticity:**
   - **Auto Scaling:** AWS provides Auto Scaling services that automatically adjust the capacity of your applications based on demand. This ensures that your application can handle sudden spikes in traffic and scale down during periods of low activity.
   - **Elastic Load Balancing:** Distributes incoming traffic across multiple instances to ensure that no single instance becomes a bottleneck.

2. **Global Reach:**
   - **Global Infrastructure:** AWS has data centers in multiple geographic regions worldwide. This allows you to deploy your application closer to your end-users, reducing latency and improving user experience.

3. **Pay-as-You-Go Pricing:**
   - **Cost Efficiency:** AWS follows a pay-as-you-go pricing model, where you pay only for the resources you consume. This is particularly advantageous for startups and businesses with uncertain growth, as it eliminates the need for large upfront capital expenditures.

4. **Managed Services:**
   - **Fully Managed Services:** AWS offers a wide range of managed services, including databases, machine learning, analytics, and more. This allows developers to focus on building applications without worrying about the underlying infrastructure.
   - **Serverless Computing:** AWS Lambda allows you to run code without provisioning or managing servers. This serverless architecture can significantly reduce operational overhead.

5. **Security and Compliance:**
   - **Comprehensive Security Model:** AWS provides a robust and secure infrastructure, with features like identity and access management (IAM), encryption, and network security. AWS is compliant with various industry standards and certifications.
   - **Shared Responsibility Model:** AWS follows a shared responsibility model, where AWS manages the security of the cloud infrastructure, and customers are responsible for securing their data and applications.

6. **Flexibility and Choice:**
   - **Diverse Service Offerings:** AWS offers a vast array of services, including computing power, storage options, databases, machine learning, and more. This allows you to choose the right combination of services that best fit your application's requirements.
   - **Open Ecosystem:** AWS supports a wide range of programming languages, frameworks, and operating systems, providing flexibility for developers to use the tools and technologies they are familiar with.

7. **DevOps Integration:**
   - **DevOps Support:** AWS provides tools and services that facilitate DevOps practices, such as continuous integration, continuous deployment, and infrastructure as code (IaC). This helps streamline development and operations processes.

8. **Reliability and Availability:**

- **High Availability:** AWS services are designed for high availability, with redundancy built into the infrastructure. This minimizes the risk of downtime and ensures that your application remains accessible.

9. **Innovation and Agility:**
   - **Rapid Innovation:** AWS continuously introduces new services and features, enabling businesses to leverage the latest technologies and stay ahead of the competition.
   - **Agility:** With the ability to provision resources quickly and scale on demand, AWS enables businesses to respond rapidly to changing market conditions and customer demands.

In summary, AWS's scalability, global reach, pay-as-you-go pricing, managed services, security measures, flexibility, and focus on innovation make it a compelling choice for web applications facing uncertain but potentially explosive growth. It allows businesses to scale seamlessly, reduce costs, and leverage a vast ecosystem of services without the complexity of owning and managing their own infrastructure.

Q8: differentiate between SOAP and REST. Which one is being used for APIs and why?

SOAP (Simple Object Access Protocol) and REST (Representational State Transfer) are two different architectural styles for designing web services. Here are the key differences between SOAP and REST:

SOAP (Simple Object Access Protocol):

1. **Protocol Type:**
   - SOAP is a protocol, a set of rules for structuring messages that are exchanged between computers.

2. **Message Format:**
   - SOAP messages are XML-based and can be quite verbose, containing a lot of metadata.
   - Typically, SOAP messages are sent over HTTP or SMTP.

3. **Statefulness:**
   - SOAP can be stateful, meaning the server can store information about the client between requests.

4. **Operations:**
   - SOAP supports a wide range of operations, including complex transactions and security.

5. **Standards:**
   - SOAP has a set of formal standards and specifications (WS-Security, WS-Transaction, etc.).

6. **Flexibility:**
   - Due to its complexity and standards, SOAP is often considered more rigid and less flexible than REST.

REST (Representational State Transfer):

1. **Architectural Style:**
   - REST is an architectural style that uses a stateless, client-server communication model.

2. **Message Format:**
   - RESTful APIs use lightweight data formats such as JSON for data exchange.
   - Data is often represented in a resource-oriented way.

3. **Statelessness:**

- REST is stateless, meaning each request from a client contains all the information needed to understand and fulfill the request.

4. **Operations:**
   - RESTful APIs are typically limited to a small set of operations (HTTP methods: GET, POST, PUT, DELETE).
   - It follows the principles of CRUD (Create, Read, Update, Delete).

5. **Standards:**
   - REST relies on standards like HTTP, URI, and MIME.

6. **Flexibility:**
   - REST is considered more flexible and scalable. It is simpler and easier to use for a wide range of clients.

Usage for APIs:

REST is more commonly used for modern web APIs for several reasons:

1. **Simplicity:**
   - REST is simpler to implement and understand. It uses standard HTTP methods and is based on the principles of the web.

2. **Flexibility:**
   - REST is more flexible and can be used with different data formats (JSON, XML) and can be easily consumed by a wide variety of clients, including browsers, mobile devices, and other applications.

3. **Scalability:**
   - Due to its stateless nature and simplicity, RESTful APIs are often more scalable and easier to maintain.

4. **Performance:**
   - JSON, commonly used with REST, is more lightweight than XML, making it faster to parse and transmit.

5. **Industry Adoption:**
   - REST has gained widespread industry adoption, and many modern frameworks and tools are built with RESTful principles in mind.

While SOAP is still used in certain scenarios, especially in enterprise environments where there may be existing SOAP-based systems, REST has become the preferred choice for building APIs in the context of modern web development due to its simplicity, flexibility, and ease of use.

Q9: develop rest API for the bookstore application. API should be created to enable the user to retrieve the book details update a new book entry and search for an entry based on a book id. Assume bookstore has book id, name, author and price

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;

// Sample in-memory database for demonstration purposes
let books = [
  { id: 1, name: 'Book1', author: 'Author1', price: 19.99 },
```

```
  { id: 2, name: 'Book2', author: 'Author2', price: 29.99 },
];

app.use(bodyParser.json());

// Get all books
app.get('/books', (req, res) => {
  res.json(books);
});
// Get a book by ID
app.get('/books/:id', (req, res) => {
  const bookId = parseInt(req.params.id);
  const book = books.find((b) => b.id === bookId);
  if (book) {
    res.json(book);
  } else {
    res.status(404).json({ message: 'Book not found' });
  }
});
// Add a new book
app.post('/books', (req, res) => {
  const { name, author, price } = req.body;
  const newBook = {
    id: books.length + 1,
    name,
    author,
    price,
  };
  books.push(newBook);
  res.status(201).json(newBook);
});
// Update a book by ID
app.put('/books/:id', (req, res) => {
  const bookId = parseInt(req.params.id);
  const index = books.findIndex((b) => b.id === bookId);
  if (index !== -1) {
    const { name, author, price } = req.body;
    books[index] = { id: bookId, name, author, price };
    res.json(books[index]);
  } else {
    res.status(404).json({ message: 'Book not found' });
  }
});

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