Part 1: React JS

1. React Lifecycle

Q1: What is the React component lifecycle?
A1: The React lifecycle is the series of methods invoked in a component's life — from creation (mounting) to updates (updating) and removal (unmounting).

Q2: What are the three main phases of the lifecycle?
A2: Mounting, Updating, and Unmounting.

Q3: Which lifecycle method is used to fetch data after a component mounts?
A3: `componentDidMount()` for class components or `useEffect()` for functional components.

Q4: What is the difference between `componentWillUnmount` and `componentDidMount`?
A4: `componentDidMount` runs after the component is added to the DOM, while `componentWillUnmount` runs before it's removed — often used for cleanup.

Q5: How do functional components handle lifecycle events?
A5: They use the `useEffect()` hook, which can replicate mounting, updating, and unmounting behaviors.

Q6: Why is `shouldComponentUpdate` important?
A6: It determines if a component should re-render, improving performance by avoiding unnecessary updates.

2. Routing

Q1: What library is commonly used for routing in React?
A1: `react-router-dom`.

Q2: How do you create a route in React Router?
A2: Use the `<Route>` component with a `path` and `element` prop, e.g., `<Route path="/home" element={<Home />} />`.

Q3: What is the difference between `<Link>` and `<a>` tags in React Router?
A3: `<Link>` prevents page reload and enables client-side navigation; `<a>` causes a full page reload.

Q4: How do you redirect in React Router v6?
A4: Use the `useNavigate()` hook to programmatically navigate to a different route.

Q5: What is a dynamic route?
A5: A route with parameters, e.g., `/user/:id`, where `id` can change.

Q6: How can you implement nested routing?
A6: Define child routes inside a parent `<Route>` and use `<Outlet />` in the parent component.

3. State Management

Q1: What is state in React?
A1: State is an object that stores dynamic data that affects how a component renders.

Q2: How do you update state in a functional component?
A2: Using the `useState()` hook and its setter function.

Q3: What is the difference between local state and global state?
A3: Local state is managed within a single component; global state is shared across components.

Q4: Name two libraries for global state management.
A4: Redux and Zustand.

Q5: Why should you never modify state directly?
A5: It can cause unexpected behavior and prevent re-rendering — always use the setter method.

Q6: How does Redux work in React?
A6: Redux stores the application state in a single store, updated via dispatched actions handled by reducers.

4. Forms in React

Q1: What is a controlled component in React forms?
A1: A component where form data is managed by React state.

Q2: How do you handle form input changes in React?
A2: Use an `onChange` handler that updates state based on `event.target.value`.

Q3: What is an uncontrolled component?
A3: A form component that stores its data in the DOM instead of React state.

Q4: How do you prevent a form from reloading the page?
A4: Call `event.preventDefault()` inside the submit handler.

Q5: How do you handle multiple inputs in a form?
A5: Use a single state object and update it using the `name` attribute of inputs.

Q6: How can you validate form data in React?
A6: Either manually in state handlers or by using libraries like `Formik` or `react-hook-form`.

5. React Authentication

Q1: How can you store authentication tokens in React?
A1: In `localStorage`, `sessionStorage`, or cookies.

Q2: What is JWT authentication?
A2: JSON Web Tokens are compact tokens used to securely transmit data between client and server.

Q3: How do you protect routes in React?
A3: By creating a PrivateRoute component that checks authentication before rendering.

Q4: How do you handle login state in React?
A4: Store user authentication data in state or a global store like Redux.

Q5: Why is it risky to store JWT in localStorage?
A5: It is vulnerable to XSS attacks — cookies with `HttpOnly` are safer.

Q6: How can you persist authentication across page reloads?
A6: Retrieve tokens from storage when the app loads and store them in state.

Part 2: MongoDB

1. CRUD Operations

Q1: What does CRUD stand for in MongoDB?
A1: Create, Read, Update, Delete — the basic operations for managing data.

Q2: How do you insert a document in MongoDB?
A2: Using `insertOne()` for a single document or `insertMany()` for multiple documents.

Q3: How do you retrieve all documents from a collection?
A3: Use `find()` without parameters, e.g., `db.collection.find({})`.

Q4: How do you update a document in MongoDB?
A4: Use `updateOne()` or `updateMany()` with `$set` to modify specific fields.

Q5: How do you delete a document in MongoDB?
A5: Use `deleteOne()` for a single document or `deleteMany()` for multiple.

---

2. Querying

Q1: How do you find documents that match specific criteria?
A1: Pass a query object to `find()`, e.g., `db.users.find({ age: { $gt: 18 } })`.

Q2: What does `$in` do in a query?
A2: Matches any value from a specified array, e.g., `{ status: { $in: ["active", "pending"] } }`.

Q3: How do you sort query results in MongoDB?
A3: Use `sort()` with 1 for ascending or -1 for descending order.

Q4: How do you limit query results?
A4: Use `limit()` to specify the maximum number of documents returned.

Q5: What does `$regex` do in MongoDB?
A5: Matches documents based on a regular expression pattern.

3. Integration with Node.js

Q1: What package is commonly used to connect Node.js with MongoDB?
A1: The official `mongodb` driver or Mongoose ODM.

Q2: How do you connect to MongoDB using Mongoose?
A2:

```
mongoose.connect("mongodb://localhost:27017/dbname", { useNewUrlParser: true, useUnifiedTopology: true });
```

Q3: How do you define a schema in Mongoose?
A3:

```
const schema = new mongoose.Schema({ name: String, age: Number });
```

Q4: How do you create a new document in Mongoose?
A4:

```
const user = new User({ name: "John", age: 30 });

await user.save();
```

Q5: How do you handle connection errors in Mongoose?
A5: Use `.catch()` or listen for the `error` event on the connection.

4. Indexing in MongoDB

Q1: What is an index in MongoDB?
A1: A data structure that improves query performance by reducing scan time.

Q2: How do you create an index?
A2: Use `createIndex()`, e.g., `db.users.createIndex({ name: 1 })`.

Q3: What is a compound index?
A3: An index on multiple fields, e.g., `{ name: 1, age: -1 }`.

Q4: How do indexes affect write operations?
A4: They can slow down writes because indexes must be updated whenever documents change.

Q5: How do you view existing indexes on a collection?
A5: Use `getIndexes()`, e.g., `db.collection.getIndexes()`.

Part 3: Node.js

1. Basic Server

Q1: What is Node.js?
A1: Node.js is a JavaScript runtime built on Chrome's V8 engine, allowing JavaScript to run outside the browser.

Q2: How do you create a basic HTTP server in Node.js?
A2:

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

http.createServer((req, res) => {

  res.write('Hello World');

  res.end();

}).listen(3000);
```

Q3: What port does a Node.js server listen on by default?
A3: There is no default port — you specify it in the `listen()` method.

Q4: How do you send a JSON response in Node.js?
A4:

```
res.writeHead(200, { 'Content-Type': 'application/json' });

res.end(JSON.stringify({ message: 'Hello' }));
```

Q5: How can you restart a Node.js server automatically on changes?
A5: Use tools like `nodemon` to watch files and restart automatically.

2. File System

Q1: How do you read a file in Node.js?
A1: Using `fs.readFile()` for asynchronous or `fs.readFileSync()` for synchronous reading.

Q2: How do you write to a file in Node.js?
A2: Use `fs.writeFile()` or `fs.appendFile()` for appending data.

Q3: How do you check if a file exists?
A3: Use `fs.existsSync()` or `fs.access()`.

Q4: How do you delete a file?
A4: Use `fs.unlink()` or `fs.unlinkSync()`.

Q5: How can you create a directory in Node.js?
A5: Use `fs.mkdir()` or `fs.mkdirSync()`.

3. Middleware

Q1: What is middleware in Node.js?
A1: Middleware is code that runs between a request and response, often used in frameworks like Express.

Q2: Can you have middleware in pure Node.js without Express?
A2: Yes, you can manually implement request/response processing logic.

Q3: What is the difference between application-level and route-level middleware?
A3: Application-level applies to all routes, route-level applies only to specific routes.

Q4: How do you pass control to the next middleware function?
A4: Call `next()` inside the middleware function.

Q5: Why is middleware important?
A5: It helps organize code for tasks like authentication, logging, and error handling.

4. Single Threaded Application

Q1: Is Node.js single-threaded?
A1: Yes, Node.js uses a single-threaded event loop for handling requests.

Q2: How does Node.js handle concurrent requests if it's single-threaded?
A2: By using asynchronous, non-blocking I/O operations via the event loop.

Q3: What module allows multi-threading in Node.js?
A3: The `worker_threads` module.

Q4: Why is Node.js suitable for I/O-heavy applications?
A4: Because it can handle many connections without blocking, thanks to non-blocking I/O.

Q5: What is the main disadvantage of Node.js being single-threaded?
A5: CPU-intensive tasks can block the event loop and slow down performance.

Great — now let's finish with **Part 4: Express.js**.

Part 4: Express.js

1. Request and Response

Q1: What is Express.js?
A1: Express.js is a minimal and flexible Node.js web application framework that provides robust features for building web and API applications.

Q2: How do you access query parameters in Express?
A2: Using `req.query`, e.g., `req.query.id`.

Q3: How do you access route parameters in Express?
A3: Using `req.params`, e.g., for `/user/:id`, access `req.params.id`.

Q4: How do you send a response in Express?
A4: Using `res.send()`, `res.json()`, or `res.status().send()`.

Q5: How do you parse incoming JSON requests?
A5: Use `express.json()` middleware, e.g., `app.use(express.json())`.

---

2. Error Handling

Q1: How do you handle errors in Express?
A1: By creating an error-handling middleware with four parameters: `(err, req, res, next)`.

Q2: How do you send a 404 response for unknown routes?
A2: Add a middleware at the end:

```
app.use((req, res) => {

  res.status(404).send('Not Found');

});
```

Q3: What is the purpose of `next(err)`?
A3: It passes the error to the next error-handling middleware.

Q4: How can you create a custom error class in Express?
A4: Extend the `Error` class and add custom properties like status code.

Q5: Can you handle async errors in Express?
A5: Yes, by using `try/catch` blocks or passing errors to `next(err)` in async functions.

3. Connecting with MongoDB

Q1: How do you connect Express to MongoDB?
A1: Using Mongoose or the MongoDB driver, e.g.,

```
mongoose.connect('mongodb://localhost:27017/dbname');
```

Q2: How do you define a model in Express with Mongoose?
A2:

```
const User = mongoose.model('User', new mongoose.Schema({ name: String }));
```

Q3: How do you create a new document from an Express route?
A3:

```
app.post('/users', async (req, res) => {
  const user = new User(req.body);
  await user.save();
  res.json(user);
});
```

Q4: How do you fetch data from MongoDB in Express?
A4:

```
app.get('/users', async (req, res) => {
  const users = await User.find();
  res.json(users);
```

});

Q5: How do you handle MongoDB connection errors in Express?
A5: Listen for `error` events on the Mongoose connection and use try/catch for async operations.

4. Scaffolding

Q1: What is scaffolding in Express.js?
A1: Scaffolding is generating boilerplate code for a project structure quickly.

Q2: How do you create an Express project using scaffolding?
A2: Use `express-generator`:

```
npx express-generator myapp
```

Q3: What does `express-generator` provide?
A3: A pre-configured folder structure with routes, views, public files, and basic middleware.

Q4: How do you install dependencies after scaffolding?
A4: Run `npm install` inside the generated project folder.

Q5: How do you start a scaffolding-based Express app?
A5: Run `npm start` or `node bin/www` in the project directory.