FSD Module 4 – 10 Marks Question Bank Answers

1. State Initialization and Asynchronous Pitfalls in React

In React, state represents the dynamic data of a component. For class components, state is initialized in the constructor using `this.state = {}`. In functional components, the `useState` hook is used:

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

Asynchronous Pitfalls:
- `useState` initialization is synchronous, but if initial values depend on asynchronous data (e.g., API calls), they won't be available immediately.
- For example:

```jsx
const [user, setUser] = useState(null);
// async fetch
useEffect(() => {
  fetchUser().then(data => setUser(data));
}, []);
```

Here, the state is updated after component mount, which may lead to a blank state during the initial render.

Solutions:
- Use `useEffect` for asynchronous tasks.
- Show loading indicators until data is available.
- Use a lazy initializer:

```jsx
useState(() => computeInitialValue());
```

Managing state with async logic carefully ensures the UI remains responsive and avoids bugs like `undefined` errors on render.

2. Lifting State Up in React

Lifting state up is the process of moving state to a common parent so that sibling components can share and manage that state consistently.

Why Necessary?
- Prevents state duplication across components.
- Enables controlled data flow from parent to children via props.
- Essential in forms, filtering, and interaction between components.

Example:

```jsx
function Parent() {
 const [name, setName] = useState("");
 return (
  <>
   <InputComponent name={name} setName={setName} />
   <DisplayComponent name={name} />
  </>
 );
}
```

Here, both child components rely on the parent for shared state.

Benefits:
- Centralized state management
- Better synchronization between components
- Easier debugging and maintenance

3. Updating State and Best Practices

In Class Components:
Use `this.setState`:

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

In Functional Components:
Use `useState`:

```jsx
```

```
setCount(prev => prev + 1);
```

Best Practices:
- Never mutate state directly. Instead, use setters.
- Use functional updates when new state depends on old state.
- Use `useEffect` for side effects (like API calls or DOM manipulation).

Complex State:
For complex state (objects, arrays), use `useReducer`:

```jsx
const [state, dispatch] = useReducer(reducer, initialState);
```

Side Effects:
Always clean up effects:

```jsx
useEffect(() => {
  const timer = setInterval(...);
  return () => clearInterval(timer);
}, []);
```

4. Component Communication in React

React supports several ways for components to communicate:

1. Props (Parent to Child):
```jsx
<Child name="React" />
```

2. Callbacks (Child to Parent):
```jsx
<Child onClick={() => setValue("clicked")} />
```

3. Context API (Across Tree Levels):
Used for global state like themes or user authentication.

```jsx
```

```
const ThemeContext = createContext();
```

4. State Management Libraries (Redux, MobX):
Used for large applications to share state across many components.

Trade-offs:

Method	Pros	Cons
Props	Simple, clear flow	Prop drilling in deep trees
Context	Avoids prop drilling	Not suited for frequent updates
Redux/MobX	Scalable for large apps	More boilerplate

5. Designing Scalable React Components

Key principles for scalable React apps:

1. Separation of Concerns:
- Divide UI into presentational and container components.

2. Reusable Components:
- Design generic components using `props.children` and dynamic props.

3. Hooks:
- Custom hooks encapsulate logic and promote reusability:

```jsx
function useForm(initial) { ... }
```

4. HOCs:
- Higher Order Components wrap and enhance existing components:

```jsx
const Authenticated = withAuth(Dashboard);
```

5. Context API:
- Share global data (theme, user) efficiently.

Folder Structure Example:
```
```

/components
/hooks
/contexts
/pages
```

Modular design leads to better testing, maintenance, and scalability.

## 6. Managing Side Effects Using useEffect

Side effects like fetching data, subscriptions, timers must be managed properly.

useEffect Hook:

```jsx
useEffect(() => {
  fetchData();
  return () => cleanup();
}, []);
```

Challenges:
- Race conditions in multiple async requests
- Memory leaks from uncleaned subscriptions

Solutions:
- Use cleanup functions
- Use AbortController to cancel requests
- Libraries like:
  - Redux Saga: Handles async logic outside of UI
  - React Query: Efficient caching and background fetching

Managing side effects well ensures a responsive and bug-free UI.

7. Express.js for RESTful APIs

Express.js is a Node.js web framework for building APIs.

How it works:
- Handles HTTP methods: GET, POST, PUT, DELETE

```js
app.get('/users', (req, res) => res.send(users));
```
```

Best Practices:
1. Separate routes into files
2. Use middleware for logging, authentication, error handling
3. Use express.Router() for modular routes

Middleware Example:

```js
app.use((req, res, next) => {
  console.log(req.url);
  next();
});
```

Express makes API building efficient through its lightweight and unopinionated design.

8. REST vs GraphQL

Feature	REST	GraphQL
Querying	Multiple endpoints	Single endpoint
Overfetching	Common	Avoided via specific queries
Flexibility	Less (fixed endpoints)	More (client-defined queries)
Versioning	Required	Avoided (query-based evolution)

Use REST when:
- Simple CRUD operations
- Caching with HTTP is important

Use GraphQL when:
- Need flexible, client-driven queries
- Minimizing API calls is critical

GraphQL is more powerful but requires schema design and query learning.

9. Strongly Typed GraphQL APIs

GraphQL uses Schema Definition Language (SDL) to define types:

```graphql
type User {
  id: ID!
  name: String!
```

```
  email: String!
}
```

Benefits:
- Type Safety: Prevents invalid queries
- Auto-documentation: Tools like GraphiQL use schema
- Better tooling: Codegen, validation

Strong typing leads to fewer bugs, faster development, and better IDE support.

10. GraphQL Query Variables

Query variables let clients pass dynamic data to queries:

Without variables:

```graphql
query {
  user(id: "123") { name }
}
```

With variables:

```graphql
query GetUser($id: ID!) {
  user(id: $id) { name }
}
```

Variables:
```json
{ "id": "123" }
```

Benefits:
- Reusable queries
- Avoids hardcoding
- Improves readability and testing

Query variables are essential for building dynamic and scalable GraphQL APIs.