IA 2 INP Question Bank

Module 4

1. Discuss Flux Architecture in detail.
2. Discuss MVC Arch. with respect to react.
3. Discuss react routing in detail.
4. Discuss react ref.

Module 5

1. Discuss the importance of Node JS in web application development with all Node JS features.
2. Discuss various Node JS modules.
3. Explain Node JS http module with suitable example.
4. Explain file system module in Node JS with example.
5. Write a Node JS program for following:
   a. Create a new file and add data into it.
   b. Append more data in the same file at the end of existing data.
   c. Read the file data without getting the buffer data.
   d. Rename the file.
   e. Delete the file.
6. Discuss Buffers and streams in Node JS with examples.
7. Write a short node on REPL.

Module 6

1. Discuss the role Express JS in web development.
2. Why Express play important role in middleware, Discuss it in details.
3. Implement Hello World Express JS program, discuss all steps.
4. Discuss routing in Express JS with detailed examples.
5. What is REST API. What are the principals of REST API.
6. Implement REST API for Student Management System.
7. Discuss Generator, Authentication, sessions with respect to express.

Module 4 (Answers)

Q) Discuss Flux Architecture in detail.

Flux is an architecture pattern introduced by Facebook for building client-side web applications. It is used to manage unidirectional data flow in React applications.
In traditional MVC patterns, data can flow in multiple directions, which makes debugging and managing state complex.
Flux ensures that data flows in one direction only, making the app predictable and easier to maintain.

Dispatcher:

- The central hub that manages all data flow.
- It receives actions and dispatches them to the stores.

Stores:

- Hold the application state and business logic.
- They listen to actions from the dispatcher and update themselves accordingly.
- When data changes, stores emit a "change" event to notify views.

Actions:

- Simple objects that contain a type and some data (payload).
- Represent user interactions or events, e.g., { type: 'ADD_TODO', text: 'Buy milk' }.

View (React Components):

- Render UI based on data from stores.
- Can trigger actions (through user input) that go back to the dispatcher.

Flux Data Flow:
User Interaction → Action → Dispatcher → Store → View (Re-render)

Q) Discuss MVC Arch. with respect to react.

MVC stands for **Model-View-Controller** — a software design pattern used for organizing code.

- **Model** → Manages data and business logic (Data layer (state, Redux, or backend API).).
- **View** → Displays data to the user (UI).
- **Controller** → Handles user input and updates Model or View (Event handlers or functions that update the state).

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

// Controller (function)
function increment() {
  setCount(count + 1);
}

// View (UI)
return <button onClick={increment}>Count: {count}</button>;
```

Here:

- State = **Model**
- increment() = **Controller**
- JSX = **View**

Q) Discuss react routing in detail.

React by itself is a Single Page Application (SPA) library, meaning it doesn't reload pages on navigation.
React Router is a library that provides client-side routing, allowing different views to be rendered without a full page refresh.

Key Features:

- Dynamic routing (URL changes without reloading the page).
- Nested routes.
- Route parameters and query strings.

- Navigation guards (using loaders or actions in v6.4+).

Main Components (React Router v6)

- BrowserRouter: Wraps the entire app; enables navigation features.
- Routes: Holds all route definitions.
- Route: Defines a path and its corresponding component.
- Link or NavLink: For navigation between routes (without reloading).
- useNavigate() Hook: For programmatic navigation.

```
import { BrowserRouter, Routes, Route, Link } from "react-router-
dom";

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
      </nav>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}

function Home() { return <h2>Home Page</h2>; }
function About() { return <h2>About Page</h2>; }
```

Benefits:

- Improves performance (no page reloads).
- Provides clean and structured navigation.
- Easy URL-based rendering of components.

Q) Discuss react ref.

Refs (short for **references**) in React are used to directly **access DOM elements or React components** without using state.

Used to:

- To **access or modify** DOM elements directly (e.g., focusing an input).
- To **store mutable values** that don't cause re-renders.
- To interact with **third-party libraries** (like animations or charts).

```
import { useRef } from "react";

function InputFocus() {
const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Focus Input</button>
    </>
  );
}
```

- useRef() creates a reference object.
- inputRef.current points to the DOM node.
- Clicking the button calls .focus() on that node.
- **Callback Refs** – A function-based alternative to useRef().
- **Forward Refs** – Used to pass refs from parent to child components using React.forwardRef().

Module 5 (Answers)

Q) Discuss the importance of Node JS in web application development with all Node JS features.

Importance of Node.js:

- JavaScript Everywhere: Allows using JavaScript for both frontend and backend
- Non-blocking I/O: Handles multiple requests simultaneously without blocking
- Single-threaded Event Loop: Efficiently manages concurrent connections
- Fast Performance: Built on Chrome's V8 JavaScript engine
- Large Ecosystem: NPM (Node Package Manager) with vast library collection
- Real-time Applications: Perfect for chat apps, gaming, live updates
- Microservices Architecture: Lightweight and scalable for microservices

Key Features:

- Event-driven architecture
- Asynchronous programming
- Cross-platform compatibility
- Fast data streaming
- No buffering
- Community support

Q) Discuss various Node JS modules.

Core Modules:

- http: Create HTTP servers and clients
- fs: File system operations
- path: Handle file paths
- os: Operating system information
- events: Event handling
- stream: Handle streaming data
- util: Utility functions
- url: URL parsing and formatting
- querystring: Query string handling

- crypto: Cryptographic functions
  Third-party Modules:
- express: Web framework
- mongoose: MongoDB object modeling
- socket.io: Real-time communication
- axios: HTTP client
- nodemon: Development tool

Q) Explain Node JS http module with suitable example.

The Node.js http module creates HTTP servers and handles web requests. It enables building web servers without external software.

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

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Yamete kudasai, onii-chan!');
});

server.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

This creates a server that responds "Yamete kudasai, onii-chan!" to all requests, demonstrating basic HTTP handling capabilities.

Q) Explain file system module in Node JS with example.

The Node.js fs module handles file operations. It provides methods for reading, writing, updating, and deleting files synchronously and asynchronously.

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

fs.readFile('file.txt', 'utf8', (err, data) => {
```

```
  if (err) throw err;
  console.log(data);
});
```

This reads a file's content without buffer data, demonstrating basic file system operations.

Q) Write a Node JS program for following:

Create a new file and add data into it:

```
const fs = require('fs');
const filename = 'example.txt';

fs.writeFile(filename, 'Initial data\n', (err) => {
  if (err) throw err;
  console.log('File created with initial data');
});
```

Append more data in the same file at the end of existing data:

```
const fs = require('fs');
const filename = 'example.txt';

fs.appendFile(filename, 'Appended data\n', (err) => {
  if (err) throw err;
  console.log('Data appended successfully');
});
```

Read the file data without getting the buffer data:

```
const fs = require('fs');
const filename = 'example.txt';

fs.readFile(filename, 'utf8', (err, data) => {
  if (err) throw err;
  console.log('File content:', data);
});
```

Rename the file:

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

fs.rename('example.txt', 'renamed_example.txt', (err) => {
  if (err) throw err;
  console.log('File renamed successfully');
});
```

Delete the file:

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

fs.unlink('renamed_example.txt', (err) => {
  if (err) throw err;
  console.log('File deleted successfully');
});
```

Q) Discuss Buffers and streams in Node JS with examples.

Q) Write a short note on REPL.

REPL is an acronym for (READ-EVAL-PRINT-LOOP). It is a method of executing JS code on the command line directly.

REPL Features:

- Interactive JavaScript environment
- Execute code immediately
- Test code snippets
- Debug and experiment

Example:

```
$ node
>
> 2 + 2
4
> const name = "JasGews911"
undefined
> console.log(Hello ${name})
Hello JasGews911
undefined
> .help  // Show all commands
> .exit  // Exit REPL
> .save filename.js  // Save session
> .load filename.js  // Load file
```

Module 6 (Answers)

Q) Discuss the role Express JS in web development.

Express.js is a minimal, flexible Node.js web framework that simplifies web application development. It serves as the **backbone for building web servers and APIs** efficiently.

Middleware Handling:

- Authentication

- Body parsing
- Error handling
- Static file serving

API Development:

- RESTful API creation
- JSON response handling
- HTTP method support (GET, POST, PUT, DELETE)

Template Rendering:

- Supports multiple view engines
- Dynamic HTML generation

Advantages:

- Minimal & Fast: Lightweight framework
- Flexible: Extensible with middleware
- Standardized: Industry standard for Node.js
- Full-featured: Handles routing, sessions, cookies, etc.

Express.js provides structure for web applications while maintaining Node.js flexibility, making it essential for modern web development.

Q) Why Express play important role in middleware, Discuss it in details.

Express middleware are functions that access requests and responses, executing between client calls and server responses.
They handle tasks like authentication, logging, and data parsing. This modular approach organizes code, enhances security, and adds functionality without cluttering route handlers, making applications more maintainable and scalable.

```
app.use(express.json()); // Parses JSON bodies
app.use((req, res, next) => {
  console.log(${req.method} ${req.path});
```

```
  next();
});
```

Q) Implement Hello World Express JS program, discuss all steps.

Files: package.json (created by npm) and index.js.

Initialize project and install Express:

```
mkdir express-hello
cd express-hello
npm init -y
npm install express
```

Create index.js:

```
// index.js
const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;

// middleware example: parse JSON body
app.use(express.json());

// route
app.get('/', (req, res) => {
  res.send('Hello World from Express!');
});

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

Run:

```
node index.js
```

Test or View:

```
curl http://localhost:3000/
```

Q) Discuss routing in Express JS with detailed examples.

Routing in Express maps HTTP methods and paths to handler functions. Routes can include parameters, query strings, middleware, and nested routers.

Key concepts:

- **HTTP method + path**: app.get('/users', handler)
- **Route parameters**: /users/:id captured as req.params.id
- **Query parameters**: ?page=2 available at req.query.page
- **Middleware**: functions that run before the route handler (e.g., auth, logging)
- **Router**: modular mountable route handler: const r = express.Router()
- **Route chains & arrays**: multiple handlers for same route; next() moves to next handler
- **Order matters**: first matching route wins. Use specific routes before wildcards.

```
app.get('/hello', (req, res) => res.send('GET hello'));
app.post('/hello', (req, res) => res.send('POST hello'));
```

**Q) What is REST API? What are the principals of REST API?

REST (Representational State Transfer) — architecture style for designing networked applications. A REST API uses HTTP verbs to operate on resources represented by URIs.

Core principles (constraints):

1. **Client–Server**: separation of concerns — client handles UI, server handles data/storage.

2. **Stateless**: each request from client contains all info the server needs; server doesn't keep session state between requests (note: sessions / auth tokens are common but session state on server violates strict statelessness).
3. **Cacheable**: responses must indicate if cacheable or not to improve performance.
4. **Uniform Interface**:
    - Resource-based URIs (e.g., /students/123)
    - Use HTTP methods appropriately: GET (read), POST (create), PUT/PATCH (update), DELETE (delete)
    - Use representations like JSON
5. **Layered System and abstraction**: client need not know whether it's communicating to actual server or intermediary (load balancers, proxies).
6. **Code on demand (optional)**: server can return executable code (e.g., JS), rarely used.
7. **Stateless & Resource manipulation via representations**.
8. **Use of appropriate HTTP status codes**: 200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 422 Un-processable Entity, 500 Server Error.

Best Practice:

- Use nouns in URI (not verbs): /students not /getStudents.
- Support filtering, pagination as query params: /students?page=2&limit=20&grade=10.
- Provide clear error messages in response body.

Q) Implement REST API for Student Management System.

A **REST API (Representational State Transfer)** allows communication between client and server using **HTTP methods**.
A **Student Management System** API performs CRUD (Create, Read, Update, Delete) operations on student data.

```
const express = require("express");
const app = express();
app.use(express.json());
```

```
let students = [];

// Create Student
app.post("/students", (req, res) => {
  students.push(req.body);
  res.send("Student added successfully");
});

// Read All Students
app.get("/students", (req, res) => {
  res.json(students);
});

// Read Single Student
app.get("/students/:id", (req, res) => {
  const student = students.find(s => s.id == req.params.id);
  res.json(student);
});

// Update Student
app.put("/students/:id", (req, res) => {
  const index = students.findIndex(s => s.id == req.params.id);
  students[index] = req.body;
  res.send("Student updated");
});

// Delete Student
app.delete("/students/:id", (req, res) => {
  students = students.filter(s => s.id != req.params.id);
  res.send("Student deleted");
});

app.listen(3000, () => console.log("Server running"));
```

Operation	HTTP Method	Endpoint	Description
Create	POST	/students	Add a new student
Read All	GET	/students	Get all students
Read One	GET	/students/:id	Get student by ID
Update	PUT	/students/:id	Update student info

Operation	HTTP Method	Endpoint	Description
Delete	DELETE	/students/:id	Remove a student

Q) Discuss Generator, Authentication, sessions with respect to express.

Generator:
Definition: Express Generator is a **command-line tool** used to **quickly create a skeleton (boilerplate)** for an Express application.
Purpose: It saves development time by automatically generating folder structure, basic routes, and configuration files.

```
npx express myApp
cd myApp
npm install
npm start
```

Structure Generated:
Includes folders like /routes, /views, /public, and app.js.

Authentication:
Definition: Authentication is the process of **verifying user identity** before allowing access to resources.
In Express: Usually implemented using **middleware** (like passport.js, jsonwebtoken, or custom logic).

- **Example Flow:**
  1. User logs in → credentials verified.
  2. Server creates a **token** (e.g., JWT).
  3. Token is sent with each request to verify the user.

```
const jwt = require("jsonwebtoken");
app.post("/login", (req, res) => {
  const token = jwt.sign({ user: req.body.user }, "secret");
  res.json({ token });
});
```

Sessions:

Definition: A **session** stores user data **on the server** to maintain state between HTTP requests (since HTTP is stateless).

In Express: Implemented using middleware like express-session.

```
const session = require("express-session");
app.use(session({ secret: "key", saveUninitialized: true, resave:
false }));

After logging:
req.session.user = "John";
res.send("Session started");
```

Purpose: Keeps users **logged in** across multiple pages without re-authentication.