Here are the **question-answer pairs** extracted from your content:

1. What is a Promise in JavaScript?

A Promise is a JavaScript feature that provides a structured and readable way to work with asynchronous code. It represents the eventual completion or failure of an asynchronous operation, allowing the result or error to be handled in an organized manner.

2. What are the three states of a Promise?

A Promise can be in one of the following three states:

- **Pending:** The initial state before the promise is resolved or rejected.
- **Fulfilled (Resolved):** The operation completed successfully, and the promise has a resulting value.
- **Rejected:** The operation failed, and the promise has a reason for the failure.

3. How do Promises help in avoiding callback hell?

Promises mitigate callback hell by enabling chaining through the `.then` method, making code more readable and maintainable.

Example:

```
// Without Promises (Callback Hell)
asyncOperation1((result1) => {
  asyncOperation2(result1, (result2) => {
    asyncOperation3(result2, (result3) => {
      // ...
    });
  });
});

// With Promises (Chaining)
asyncOperation1()
  .then((result1) => asyncOperation2(result1))
  .then((result2) => asyncOperation3(result2))
  .then((result3) => {
    // ...
  });
```

4. How do Promises handle errors?

Promises have built-in error handling using the `.catch` method, which allows a centralized way to handle errors in a sequence of asynchronous operations.

Example:

```
asyncOperation1()
  .then((result1) => asyncOperation2(result1))
  .then((result2) => asyncOperation3(result2))
  .catch((error) => {
    console.error('An error occurred:', error);
  });
```

5. What is `Promise.all`, and when should you use it?

`Promise.all` allows multiple promises to execute in parallel and waits for all of them to complete before proceeding. It is useful when multiple independent asynchronous operations need to be completed before moving forward.

Example:

```javascript
const promise1 = asyncOperation1();
const promise2 = asyncOperation2();

Promise.all([promise1, promise2])
  .then((results) => {
    const result1 = results[0];
    const result2 = results[1];
    // ...
  })
  .catch((error) => {
    console.error('An error occurred:', error);
  });
```

6. How do you create a Promise in JavaScript?

A Promise is created using the `Promise` constructor, which takes a function with two parameters: `resolve` and `reject`.

Example:

```javascript
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation
  if (/* success */) {
    resolve('Operation succeeded!');
  } else {
    reject('Operation failed!');
  }
});
```

7. How do you resolve a Promise?

A Promise is resolved by calling the `resolve` function inside its executor function.

Example:

```
const successfulPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Operation succeeded!');
  }, 1000);
});
```

8. How do you reject a Promise?

A Promise is rejected by calling the `reject` function inside its executor function.

Example:

```
const failedPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject('Operation failed!');
  }, 1000);
});
```

9. How do you consume a Promise?

A Promise can be consumed using `.then` for successful resolution and `.catch` for error handling.

Example:

```
successfulPromise
  .then((result) => {
    console.log(result); // Output: Operation succeeded!
  })
  .catch((error) => {
    console.error(error);
  });
```

10. What happens when you chain `.then()` methods on a Promise?

Each `.then()` returns a new Promise, allowing chaining of multiple asynchronous operations sequentially.

Example:

```
successfulPromise
  .then((result) => {
    console.log(result); // Output: Operation succeeded!
    return 'New value';
  })
  .then((newValue) => {
    console.log(newValue); // Output: New value
  })
  .catch((error) => {
    console.error(error);
  });
```

11. How does `Promise.all` handle errors?

If any of the promises in `Promise.all` fail, the entire `Promise.all` operation fails, and the `.catch()` block is executed.

Example:

```
const promise1 = Promise.resolve('One');
const promise2 = Promise.reject('Error in promise2');

Promise.all([promise1, promise2])
  .then((values) => {
    console.log(values);
  })
  .catch((error) => {
    console.error('An error occurred:', error); // Output: Error in promise2
  });
```

This set of **Q&A** covers the **key aspects** of Promises while keeping the answers strictly from your content. 🚀 Let me know if you need modifications!

Express Server - Question & Answer Format

1. What is Express, and what is its purpose?

Answer:
Express is a web application framework for Node.js, designed to simplify the process of building web applications and APIs.

---

2. What are the common HTTP methods used in Express?

Answer:

- **GET:** Used to retrieve data from the server.
- **POST:** Used to submit data to the server.
- **Other Methods (PUT, DELETE, etc.):** Used for updating or deleting resources.

---

3. What is a route in Express?

Answer:
A route defines the paths in an Express application and specifies how the application should respond to different HTTP requests.

---

4. How do you define a basic route in Express?

Answer:
You can define a route using the following syntax:

```
app.get('/', (req, res) => {
  res.send('Hello, this is the root/main route!');
});
```

This sets up a route that responds with `"Hello, this is the root/main route!"` when accessed at `/`.

---

5. What are the `req` and `res` objects in Express?

Answer:

- `req` **(Request Object):** Represents the incoming HTTP request and contains details such as parameters, headers, and body.
- `res` **(Response Object):** Represents the HTTP response sent back to the client. It allows sending data, setting headers, and controlling the response.

---

6. How do you create a dynamic route using URL parameters in Express?

Answer:
You can define a route with parameters like this:

```
app.get('/greet/:name', (req, res) => {
  const { name } = req.params;
  res.send(`Hello, ${name}!`);
});
```

When accessed via `/greet/John`, it responds with `"Hello, John!"`.

---

7. How do you send a JSON response in Express?

Answer:

You can send a JSON response using `res.json()`:

```
app.get('/api', (req, res) => {
  res.json({ message: 'This is the API route.' });
});
```

---

8. How do you start an Express server on port 3000?

Answer:
You start the server using the `listen` method:

```
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

---

9. How do you run the Express server after writing the code?

Answer:
You run the server using the command:

```
node server.js
```

---

10. How do you access the Express server in the browser after starting it?

Answer:
Visit **`http://localhost:3000`** in your browser. You can also try:

- **`http://localhost:3000/api`** → Returns a JSON response.
- **`http://localhost:3000/greet/yourname`** → Responds with `"Hello, yourname!"`.

Nodemon - Question & Answer Format

1. What is Nodemon, and what is its purpose?

Answer:
Nodemon is a utility that monitors for changes in the source code and automatically restarts the server when changes are detected.

---

2. How do you install Nodemon globally?

Answer:
You can install Nodemon globally using the following command:

```
npm install -g nodemon
```

---

3. How do you install Nodemon locally in a project?

Answer:
You can install Nodemon as a development dependency using:

```
npm install --save-dev nodemon
```

---

4. How do you start a server using Nodemon?

Answer:
You can start the server with Nodemon using:

```
nodemon server.js
```

---

5. Why should you use Nodemon in development?

Answer:
Nodemon automatically restarts the server whenever changes are made, making the development process smoother by eliminating the need to manually restart the server after every change.

GET vs. POST - Question & Answer Format

1. What is the main difference between `GET` and `POST` methods?

Answer:

- `GET` is used to retrieve data from a server, while `POST` is used to send data to the server for processing.

---

2. How is data sent in a `GET` request?

Answer:

- In a `GET` request, data is appended to the URL as query parameters.

---

3. How is data sent in a `POST` request?

Answer:

- In a `POST` request, data is sent in the request body, not in the URL.

---

4. Why is `GET` less secure than `POST`?

Answer:

- In `GET`, parameters are visible in the URL, which can expose sensitive information.
- `POST` keeps data in the request body, making it more secure.

---

5. Can `GET` requests be cached and bookmarked?

Answer:

- Yes, `GET` requests can be cached by browsers and URLs can be bookmarked.

---

6. Can `POST` requests be cached or bookmarked?

Answer:

- No, `POST` requests are not cached, and URLs cannot be bookmarked.

---

7. Which HTTP method should be used for retrieving data?

Answer:

- `GET` should be used when retrieving data from the server.

---

8. Which HTTP method should be used for submitting forms or creating resources?

Answer:

- `POST` should be used for submitting forms and creating resources on the server.

---

9. Is `GET` idempotent?

Answer:

- Yes, multiple identical `GET` requests will have the same effect as a single request.

---

10. Is `POST` idempotent?

Answer:

- No, multiple `POST` requests may create multiple resources or trigger multiple actions.

---

11. Give an example of when to use `GET`.

Answer:

- Searching on Google or fetching a blog post from a website.

---

12. Give an example of when to use `POST`.

Answer:

- Submitting a login form, uploading a file, or making an online payment.

Handling GET Requests - Question & Answer Format

1. How do you create an Express application?

Answer:

- You create an Express application using `express()` and store it in a variable, e.g., `const app = express();`.

---

2. How do you define a simple GET route in Express?

Answer:

- Use `app.get('/', (req, res) => { res.send('Hello, this is a GET request!'); });`.

---

3. What does `req.params` do in a GET route?

Answer:

- `req.params` allows access to route parameters, which can be used to handle dynamic values in the URL.

---

4. How do you define a GET route with a parameter?

Answer:

- Use `app.get('/greet/:name', (req, res) => { const { name } = req.params; res.send(\` Hello, ${name}!`); });`.

---

5. What happens when you visit `http://localhost:3000/greet/John`?

Answer:

- The server responds with `"Hello, John!"` because `John` is passed as a route parameter.

---

6. How do you start an Express server?

Answer:

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

---

7. What port is used in the example for running the Express server?

Answer:

- The server runs on port **3000**.

---

8. What will you see when you visit `http://localhost:3000/` in the browser?

Answer:

- The browser will display `"Hello, this is a GET request!"`.

---

9. What is the purpose of `app.listen(port, callback)` in an Express server?

Answer:

- It starts the server on the specified port and executes the callback when the server starts successfully.

---

10. What is the advantage of handling GET requests in Express?

Answer:

- Express provides a flexible and powerful routing system to handle different HTTP methods, route parameters, and query parameters easily.

Handling POST Requests - Question & Answer Format

1. Why is middleware needed for handling POST requests in Express?

Answer:

- Middleware is needed to parse the incoming request data, such as JSON and form data, before handling it in the route.

---

2. What middleware is used to parse JSON data in Express?

Answer:

- `app.use(express.json());` is used to parse incoming JSON data in requests.

---

3. What middleware is used to parse form data in Express?

Answer:

- `app.use(express.urlencoded({ extended: true }));` is used to parse URL-encoded form data.

---

4. What is the purpose of the `/add-content` POST route?

Answer:

- The `/add-content` route allows clients to send text content via a POST request, which is stored in an in-memory array.

---

5. How does the server extract data from a POST request?

Answer:

- The server extracts data using `req.body.content`, assuming the client sends JSON or form data with a `content` field.

---

6. What happens if the client sends a POST request without a `content` field?

Answer:

- The server responds with a **400 Bad Request** status and returns `{ error: 'Content is required' }`.

---

7. What status code is returned when content is successfully added?

Answer:

- The server responds with **201 Created** and `{ message: 'Content added successfully' }`.

---

8. Where is the received content stored in this example?

Answer:

- It is stored in an in-memory array called `textContent`.

---

9. Why might you need more robust validation for handling POST requests?

Answer:

- To ensure data integrity, prevent empty or invalid inputs, and handle security concerns like SQL injection and XSS attacks.

---

10. How do you send a POST request to the `/add-content` route?

Answer:

- Use tools like **Postman**, a **frontend form**, or a command-line tool like **cURL** to send a request with a JSON body:

```
{
  "content": "Sample text content"
}
```

Response Object in Express.js - Question & Answer Format

1. What is the purpose of the `res` object in Express.js?

Answer:

- The `res` object is used to send responses back to the client in different formats, such as text, JSON, HTML, files, and redirects.

---

2. How do you send a plain text response in Express?

Answer:

- Use the `res.send` method:

  ```
  app.get('/', (req, res) => {
    res.send('Hello, this is a plain text response!');
  });
  ```

---

3. How do you send a JSON response?

Answer:

- Use the `res.json` method:

  ```
  app.get('/api/data', (req, res) => {
    res.json({ message: 'This is a JSON response.' });
  });
  ```

---

4. How do you send an HTML response?

Answer:

- Use the `res.send` method with HTML content:

  ```
  app.get('/html', (req, res) => {
    res.send('<h1>This is an HTML response</h1>');
  });
  ```

---

5. How do you redirect a client to another URL?

Answer:

- Use the `res.redirect` method:

  ```
  app.get('/redirect', (req, res) => {
    res.redirect('/new-location');
  });
  ```

---

6. How do you send a 404 "Not Found" response?

Answer:

- Use the `res.status` method with `res.send`:

  ```
  app.get('/not-found', (req, res) => {
    res.status(404).send('Page not found');
  });
  ```

---

7. How do you send a file as a response?

Answer:

- Use the `res.sendFile` method:

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

app.get('/file', (req, res) => {
  const filePath = path.join(__dirname, 'files', 'example.txt');
  res.sendFile(filePath);
});
```

---

8. How do you set a custom HTTP header in a response?

Answer:

- Use the `res.set` method:

```
app.get('/custom-header', (req, res) => {
  res.set('X-Custom-Header', 'Custom Header Value');
  res.send('Response with a custom header');
});
```

---

9. What is the difference between `res.send` and `res.json`?

Answer:

- `res.send` can send **text, HTML, or JSON**, while `res.json` is specifically for **JSON responses** and automatically sets the `Content-Type` to `application/json`.

---

10. Why is `res.status` useful before sending a response?

Answer:

- It allows setting an **HTTP status code** before sending a response to indicate success (`200`), errors (`400`, `404`), or redirections (`302`).

Serving Routes in Express.js - Question & Answer Format

1. How do you define routes in an Express.js application?

Answer:

- Routes in Express are defined using the `app` object with HTTP methods (`GET`, `POST`, `PUT`, `DELETE`, etc.), specifying a URL path and a callback function to handle requests.

---

2. How do you create a `GET` route in Express.js?

Answer:

- Use `app.get()` to handle `GET` requests:

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

---

3. How do you create a `POST` route in Express.js?

Answer:

- Use `app.post()` to handle `POST` requests:

```
app.post('/add', (req, res) => {
  res.send('Hello from POST route!');
});
```

---

4. How do you create a `PUT` route for updating data?

Answer:

- Use `app.put()` and define a parameter in the URL:

```
app.put('/put/:id', (req, res) => {
  res.send('Hello from PUT route!');
});
```

---

5. How do you create a `DELETE` route for deleting data?

Answer:

- Use `app.delete()` with a URL parameter:

```
app.delete('/delete/:id', (req, res) => {
  res.send('Hello from DELETE route!');
});
```

---

6. How do you start an Express server to listen for requests?

Answer:

- Use `app.listen()` to start the server on a specific port:

```
const port = 3000;

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

---

7. What is the purpose of route parameters (`:id`) in Express.js?

Answer:

- Route parameters allow capturing dynamic values from the URL, such as an `id` for identifying specific resources (e.g., updating or deleting an item).
  - Example:

```
app.put('/put/:id', (req, res) => {
  res.send(`Updating item with ID: ${req.params.id}`);
});
```

---

8. What is the difference between `GET`, `POST`, `PUT`, and `DELETE` routes?

Answer:

- `GET` → Retrieves data from the server.
- `POST` → Sends new data to the server.
- `PUT` → Updates existing data on the server.
- `DELETE` → Removes data from the server.

---

9. How do you handle multiple routes in an Express app?

Answer:

- Define multiple `app.get()`, `app.post()`, `app.put()`, and `app.delete()` routes for different URLs in your Express application.

---

10. What happens if no route matches a request?

Answer:

- Express will return a **404 Not Found** response unless a wildcard (`*`) route or error-handling middleware is defined.
  - Example of handling unknown routes:

```
app.use((req, res) => {
  res.status(404).send('Page not found');
});
```

Understanding ENV in Express.js - Question & Answer Format

1. What is the purpose of the `dotenv` package in an Express.js application?

Answer:

- The `dotenv` package loads environment variables from a `.env` file into `process.env`, making them accessible within the application.

---

2. How do you install the `dotenv` package?

Answer:

- Run the following command in the terminal:

```
npm install dotenv
```

---

3. How do you create and configure a `.env` file?

Answer:

- Create a file named `.env` in the root of your project and define variables:

```
PORT=3000
```

---

4. How do you load environment variables in an Express.js app?

Answer:

- Add the following line at the top of your main file (`app.js` or `index.js`):

```
require('dotenv').config();
```

5. How do you use the `PORT` environment variable in an Express app?

Answer:

- Access the `PORT` variable using `process.env.PORT`, providing a default if not set:

  ```
  const port = process.env.PORT || 3000;
  ```

6. What is the complete code to start an Express.js server using an environment variable for the port?

Answer:

```
require('dotenv').config();
const express = require('express');
const app = express();
const port = process.env.PORT || 3000;

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

7. How do you run an Express.js app that uses environment variables?

Answer:

- Use the following command:

  ```
  node app.js
  ```

8. Why should you use an environment variable instead of hardcoding the port?

Answer:

- Using environment variables allows flexibility when deploying the app to different environments (development, production, etc.), avoiding conflicts with hardcoded values.

---

9. What happens if the `PORT` variable is not set in the `.env` file?

Answer:

- The app will use the default port value specified in `process.env.PORT || 3000`, so it will run on port 3000.

---

10. Should you commit the `.env` file to version control (Git)?

Answer:

- No, `.env` files should not be committed. Instead, add it to `.gitignore` to keep environment variables private.

Parsing Body in a POST Request - Question & Answer Format

1. How can you parse the body of a POST request in Express.js?

Answer:

- You can use middleware to handle different types of data formats such as JSON or form data. Express provides built-in middleware:

```
app.use(express.json()); // For parsing JSON data
app.use(express.urlencoded({ extended: true })); // For parsing form data
```

---

2. Which middleware is used to parse JSON data in Express?

Answer:

- `express.json()` is used to parse JSON data in an Express application:

  ```
  app.use(express.json());
  ```

---

3. Which middleware is used to parse form data in Express?

Answer:

- `express.urlencoded({ extended: true })` is used to parse form data in an Express application:

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

---

4. How do you handle form data in a POST request?

Answer:

- Use the following route to handle form data:

  ```
  app.post('/form', (req, res) => {
    const formData = req.body;
    res.json({ receivedData: formData });
  });
  ```

---

5. How do you handle JSON data in a POST request?

Answer:

- Use the following route to handle JSON data:

```
app.post('/json', (req, res) => {
  const jsonData = req.body;
  res.json({ receivedData: jsonData });
});
```

---

6. What is the purpose of `req.body` in a POST request?

Answer:

- `req.body` contains the parsed data sent in the request body, which can be accessed after using the appropriate middleware.

---

7. What is the default port in the provided Express.js example?

Answer:

- The default port in the example is `3000`:

  ```
  const port = 3000;
  ```

---

8. How do you start the Express server in the provided example?

Answer:

- Use the following command:

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

Parsing Headers in a GET Request - Question & Answer Format

1. How can you access headers in a GET request in Express.js?

Answer:

- You can access headers using the `req.headers` object inside a route handler.

---

2. How do you retrieve the 'User-Agent' header from a request?

Answer:

- Use the following code to extract the 'User-Agent' header:

```
const userAgent = req.headers['user-agent'];
```

---

3. How do you retrieve the 'Accept-Language' header from a request?

Answer:

- Use the following code to extract the 'Accept-Language' header:

```
const acceptLanguage = req.headers['accept-language'];
```

---

4. What is the purpose of the `req.headers` object?

Answer:

- The `req.headers` object contains all headers sent with an HTTP request and allows you to access specific headers by their names.

---

5. How do you send the extracted headers as a JSON response?

Answer:

- Use the following code to send the headers as a JSON response:

```
res.json({
  userAgent,
  acceptLanguage,
});
```

---

6. What is the default port in the provided Express.js example?

Answer:

- The default port is **3000**:

```
const port = 3000;
```

---

7. How do you start an Express.js server in the provided example?

Answer:

- Use the following code:

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

Parsing Query Parameters - Question & Answer Format

1. How can you access query parameters in an Express.js application?

Answer:

- You can access query parameters using the `req.query` object inside a route handler.

---

2. How are query parameters included in a URL?

Answer:

- Query parameters are included in the URL after the `?` character and separated by `&`.
  Example:

  ```
  https://www.example.com/api/user?id=123&name=John
  ```

---

3. How do you retrieve the `id` query parameter from a request?

Answer:

- Use the following code to extract the `id` query parameter:

  ```
  const userId = req.query.id;
  ```

---

4. How do you retrieve the `name` query parameter from a request?

Answer:

- Use the following code to extract the `name` query parameter:

  ```
  const name = req.query.name;
  ```

---

5. What is the purpose of the `req.query` object?

Answer:

- The `req.query` object contains all query parameters sent in the request URL and allows access to them by their names.

---

6. How do you send the parsed query parameters as a JSON response?

Answer:

- Use the following code to send the extracted query parameters as a JSON response:

```
res.json({ user: { id: userId, name: name } });
```

---

7. What is the default port used in the provided Express.js example?

Answer:

- The default port is **3000**:

```
const port = 3000;
```

---

8. How do you start an Express.js server in the provided example?

Answer:

- Use the following code:

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

JSON Response - Question & Answer Format

1. How do you send a JSON response in an Express.js application?

Answer:

- You can send a JSON response using the `res.json()` method.

  ```
  app.get('/api/data', (req, res) => {
    res.json({ message: 'This is a JSON response.' });
  });
  ```

---

2. What is the purpose of the `res.json()` method?

Answer:

- The `res.json()` method automatically sets the `Content-Type` header to `application/json` and sends the response in JSON format.

---

3. Does `res.json()` automatically set the appropriate `Content-Type` header?

Answer:

- Yes, `res.json()` automatically sets `Content-Type: application/json`.

---

4. How does the client receive the JSON response?

Answer:

- The client receives the response as a properly formatted JSON object.
  Example Response:

  ```
  {
    "message": "This is a JSON response."
  }
  ```

---

5. What is an example of sending dynamic JSON data in a response?

Answer:

- You can send dynamic JSON data based on request parameters or database queries.

```
app.get('/api/user', (req, res) => {
  const user = { id: 1, name: 'John Doe' };
  res.json(user);
});
```

---

6. Can `res.json()` send an array as a response?

Answer:

- Yes, `res.json()` can send an array as a JSON response.

```
app.get('/api/users', (req, res) => {
  res.json([
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' }
  ]);
});
```

---

7. What happens if you pass a non-JSON object to `res.json()`?

Answer:

- Express will attempt to serialize the data into a valid JSON format before sending the response.

---

8. How do you start an Express.js server to send a JSON response?

Answer:

- Example:

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/api/data', (req, res) => {
  res.json({ message: 'This is a JSON response.' });
});

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

Postman - Question & Answer Format

1. What is Postman?

Answer:

- Postman is a popular API development and testing tool that allows users to make HTTP requests and view responses.

---

2. Where can you download Postman?

Answer:

- You can download Postman from the official website: Postman Download.

---

3. How do you create a new request in Postman?

Answer:

- Open Postman, click on the **"New"** button at the top left, and select **"HTTP"** to create a new request.

---

4. How do you make a GET request using Postman?

Answer:

- Steps to make a GET request:
  1. Open Postman.
  2. Enter your server URL (e.g., `http://localhost:3000/`) in the request bar.
  3. Select **"GET"** as the method.
  4. Click the **"SEND"** button.
  5. View the response in the Postman interface.

---

5. What should you enter in Postman's request bar to test a locally running Express server?

Answer:

- You should enter `http://localhost:3000/` (or the appropriate port where your server is running).

---

6. What response will you get in Postman when making a successful GET request to an Express server?

Answer:

- If the server is running and responding properly, you will see the response returned by Express, such as:

```
{
  "message": "Hello from Express server!"
}
```

---

7. How do you test other HTTP methods like POST, PUT, or DELETE in Postman?

Answer:

- To test other HTTP methods:
    i. Select **POST**, **PUT**, or **DELETE** from the dropdown next to the request bar.
    ii. Enter the server URL (e.g., `http://localhost:3000/api`).
    iii. Go to the **Body** tab and enter the request payload if required.
    iv. Click **SEND** to execute the request.

---

8. Why is Postman useful for API testing?

Answer:

- Postman allows developers to:
    - Test API endpoints without writing client-side code.
    - Send different types of HTTP requests (GET, POST, PUT, DELETE).
    - View server responses and debug API issues.
    - Send request headers and body data easily.

Questions and Answers Based on the Provided Content

1. What is the purpose of middlewares in Express.js?

Answer: Middlewares in Express.js are used to perform prechecks or operations before the main application logic runs. They help in organizing code by separating preliminary checks (like authentication and input validation) from the core logic, adhering to the DRY (Don't Repeat Yourself) principle.

---

2. What is the role of the `next()` function in middleware?

Answer: The `next()` function in middleware is a callback that passes control to the next middleware in the stack. If `next()` is not called, the request-response cycle stops, and the client does not receive a response.

---

3. What is the difference between `res.send` and `res.json`?

Answer:

- `res.send` sends a response of various types (string, Buffer, object, etc.), and Express automatically sets the `Content-Type` based on the data.
- `res.json` specifically sends a JSON response and sets the `Content-Type` to `application/json`.

---

4. Why is `app.use(express.json())` important in Express?

Answer: `app.use(express.json())` is middleware that parses incoming JSON payloads in the request body. Without it, JSON data in the request body would not be automatically parsed, requiring manual handling.

---

5. What are the three ways to send inputs to an Express server?

Answer:

1. **Query Parameters** (e.g., `?key=value` in the URL).
2. **Body** (hidden part of the request, used for form submissions or JSON data).
3. **Headers** (metadata like authentication tokens or content type).

---

6. What is the purpose of a global catch (error-handling middleware)?

Answer: A global catch is an error-handling middleware that centralizes error management, ensuring consistent error responses and preventing crashes due to unhandled exceptions. It uses four arguments (`err, req, res, next`) to catch and process errors.

---

7. Why is input validation important in backend applications?

Answer: Input validation ensures that the data received by the server meets expected criteria (e.g., correct format, length, or type), preventing vulnerabilities like malformed inputs or server crashes due to invalid data.

---

8. How does Zod simplify input validation compared to manual `if-else` checks?

Answer: Zod allows developers to define schemas declaratively (e.g., `z.string().min(3)`), automating validation and reducing repetitive `if-else` blocks. It throws structured errors if validation fails, making the code cleaner and less error-prone.

---

9. What are some key features of Zod?

Answer:

- Supports basic types (`string`, `number`, etc.).
- Object and nested schema definitions.
- Array, union, and intersection types.
- Optional/nullable fields and custom validators.
- Automatic parsing and TypeScript integration.

---

10. How does Zod handle validation errors?

Answer: Zod's `parse()` method throws an error with details (e.g., `error.errors`) if validation fails, allowing developers to send structured error responses (e.g., `res.status(400).json({ error: error.errors })`).

Below is a set of questions and answers derived directly from the content you provided for **Week 3.2: Authentication & Databases**. These focus on key concepts such as the `fetch()` method, asynchronous programming, authentication strategies, and MongoDB, ensuring all answers are sourced from the lecture notes.

Questions and Answers on fetch() Method

Q1: What is the `fetch()` method in JavaScript, and what does it return?
A1: The `fetch()` method is a built-in JavaScript function that simplifies making HTTP requests to retrieve data from a server. It returns a Promise that resolves to the `Response` object representing the request's outcome, whether successful or not.

Q2: Why is the `fetch()` method used in web applications?
A2: The `fetch()` method is used for:

1. **Asynchronous Data Retrieval:** It retrieves data from a server without blocking code execution.
2. **Web API Interaction:** It simplifies requests to external APIs.
3. **Promise-Based:** It uses Promises for cleaner async handling with `.then()` and `.catch()`.
4. **Flexible and Powerful:** It offers more options than older methods like `XMLHttpRequest`.

Q3: How do you handle a successful response and errors using `fetch()`?
A3: You use `.then()` to process the response and `.catch()` for errors. Example:

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) throw new Error(`HTTP error! Status: ${response.status}`);
    return response.json();
  })
  .then(data => console.log('Data:', data))
  .catch(error => console.error('Fetch error:', error));
```

Questions and Answers on Asynchronous Concepts

Q4: What is a callback function, and how does it relate to `fetch()`?
A4: A callback function is passed as an argument to another function and executed after its completion. In older code, callbacks handled async operations, like those in `.then()` blocks of `fetch()`. Example:

```
function fetchData(callback) {
  setTimeout(() => callback('Hello, callback!'), 1000);
}
fetchData(result => console.log(result));
```

Q5: What is a Promise, and how does it connect to `fetch()`?
A5: A Promise is an object representing the eventual success or failure of an async operation, improving readability over callbacks. `fetch()` returns a Promise, handled with `.then()` for success and `.catch()` for errors. Example:

```
fetchDataPromise()
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

Q6: How does `async/await` improve asynchronous code, and how is it used with

`fetch()`?

A6: `async/await` is syntactic sugar over Promises, making async code look synchronous and more readable. With `fetch()`, it simplifies response handling. Example:

```
async function fetchDataAndPrint() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
fetchDataAndPrint();
```

Q7: What is the purpose of a `try-catch` block in JavaScript?
A7: A `try-catch` block handles runtime errors gracefully, preventing program crashes. The `try` block runs code that might fail, and `catch` handles any exceptions. Example:

```
try {
  const result = 10 / 0;
} catch (error) {
  console.error('An error occurred:', error.message);
}
```

Questions and Answers on Authentication

Q8: What is hashing, and why is it used in authentication?
A8: Hashing converts a password into a fixed-size hash value using a one-way function (e.g., bcrypt). It's used to enhance security by storing hashed passwords instead of plaintext, reducing breach risks. Example:

```
bcrypt.hash('user123', 10, (err, hash) => console.log('Hashed:', hash));
```

Q9: How does encryption differ from hashing in authentication?
A9: Encryption is a two-way process that protects data confidentiality with a reversible key, unlike hashing's one-way process. It's used for sensitive data like credit card details. Example:

```
const cipher = crypto.createCipher('aes-256-cbc', 'secretKey');
let encrypted = cipher.update('Sensitive information', 'utf-8', 'hex');
encrypted += cipher.final('hex');
```

Q10: What is a JSON Web Token (JWT), and how does it work in authentication?
A10: A JWT is a digital token with a header, payload, and signature, used to verify user identity. After login, the server issues a JWT, which the client sends with requests. The server verifies it with a key. Example structure: `Header.Payload.Signature`.

Q11: How does Local Storage support authentication?
A11: Local Storage stores authentication tokens (e.g., JWTs) client-side, persisting sessions across browser restarts. It reduces authentication overhead and improves performance by avoiding repeated logins.

Q12: What is the Authorization header, and how is it used?
A12: The Authorization header sends credentials (e.g., `Bearer <token>`) in HTTP requests to authenticate clients. The server verifies it to grant access. Example:

```
axios.get('https://api.example.com/resource', {
  headers: { 'Authorization': `Bearer ${token}` }
});
```

Q13: How do cookies differ from Local Storage for storing JWTs?
A13: Cookies are secure with HTTP-only flags, auto-sent with requests, and can expire, while Local Storage is easier to use, has more capacity, but requires manual request attachment and expiration handling.

Q14: What are the advantages of Axios over `fetch()` for HTTP requests?
A14: Axios auto-parses JSON, supports interceptors, offers detailed request configuration, and ensures consistent behavior across browsers, unlike `fetch()`, which requires manual JSON parsing and lacks interceptors.

Questions and Answers on Databases and MongoDB

Q15: Why is storing data in memory problematic for real-world applications?
A15: 1. **Data Can't Be Dynamic:** In-memory data is volatile and lost on restart.
2. **Multiple Servers:** Data isn't shared across servers, causing inconsistencies.

Q16: What are the four types of databases mentioned, and their examples?
A16: 1. **Graph Databases:** Neo4j (for relationships).
2. **Vector Databases:** InfluxDB (for spatial data).
3. **SQL Databases:** PostgreSQL (tabular structure).
4. **NoSQL Databases:** MongoDB (flexible modeling).

Q17: What are the key features of MongoDB?
A17: 1. **Database Creation:** Multiple databases as containers.
2. **Collection Creation:** Groups documents like tables.
3. **JSON Data Storage:** Uses BSON format.
4. **Schemaless Design:** Flexible document structures.

5. **Scalability:** Horizontal scaling across servers.
6. **Versatility:** Adapts to diverse use cases.

Q18: What is Mongoose, and how does it connect to MongoDB?
A18: Mongoose is an ODM library for MongoDB and Node.js, providing schema-based abstraction. It simplifies connecting to MongoDB, defining models, and performing CRUD operations.

Q19: How do Express, JWT, and Mongoose work together to connect a backend to a database?
A19: Express creates the server and handles requests. **JWT** authenticates users via tokens, reducing server-side session storage. **Mongoose** connects to MongoDB, managing data interactions with schemas and models.

Q20: What is the first step to create a free MongoDB instance using MongoDB Atlas?
A20: Visit the MongoDB Atlas website at MongoDB Atlas and sign up or log in.

Below is a set of questions and answers derived directly from the content you provided for **Week 4.1: Introduction to DOM**. These focus on the critical topics of ECMAScript, DOM manipulation, backend interaction, and debouncing, ensuring all answers are sourced from the lecture notes without adding external information.

Questions and Answers on ECMA Scripts and Auxiliary APIs

Q1: What is ECMAScript, and how does it relate to JavaScript?
A1: ECMAScript (ES) is a scripting language specification that serves as the standard upon

which JavaScript is based. It acts as a set of rules and guidelines ensuring JavaScript behaves consistently, like a manual defining its features and functionality.

Q2: What are Auxiliary APIs in web development?
A2: Auxiliary APIs are additional interfaces and functionalities provided by browsers or runtime environments beyond core JavaScript (ECMAScript). They extend JavaScript's capabilities for tasks like browser interaction, server-side operations, or third-party service integration.

Q3: What are some examples of Auxiliary APIs mentioned, and their purposes?
A3:

1. **Node.js APIs:** Provide server-side features like file system access (e.g., `fs.readFile`).
2. **Third-Party APIs:** Enable integration with external services (e.g., Google Maps API for maps).
3. **Web APIs:** Offer browser-specific functionality like DOM manipulation, Fetch API for requests, and Web Storage for local storage.

Questions and Answers on Understanding Document and DOM Manipulation

Q4: What is the Document in JavaScript, and what is its role in the DOM?
A4: The `document` object is the root object of the DOM (Document Object Model), representing the entire HTML or XML document. It provides an entry point to the DOM, allowing developers to use its methods and properties to dynamically interact with and manipulate a web page's content.

Q5: How does the DOM allow JavaScript to manipulate HTML?
A5: The DOM represents the HTML document as a tree-like structure, with each element (e.g., buttons, paragraphs) as a node. JavaScript can interact with this tree to change, add, or

remove elements, updating the webpage dynamically.

Q6: How can JavaScript change HTML content using the DOM, with an example?
A6: JavaScript uses DOM methods like `getElementById` and properties like `textContent` to modify HTML. Example:

```
<button id="simpleButton">Click me!</button>
<p id="output"></p>
<script>
  const button = document.getElementById('simpleButton');
  const para = document.getElementById('output');
  button.addEventListener('click', () => {
    para.textContent = 'Awesome! You clicked the button!';
  });
</script>
```

When clicked, the button updates the paragraph’s content.

Q7: How does DOM rendering work in a simple calculator example?
A7: In a calculator example, DOM rendering retrieves input values, calculates a result, and updates the page:

```
<input id="num1" type="number">
<input id="num2" type="number">
<button id="calculateButton">Calculate Sum</button>
<p id="sumResult"></p>
<script>
  const num1Input = document.getElementById('num1');
  const num2Input = document.getElementById('num2');
  const button = document.getElementById('calculateButton');
  const result = document.getElementById('sumResult');
  button.addEventListener('click', () => {
    const sum = (parseFloat(num1Input.value) || 0) + (parseFloat(num2Input.value) || 0);
    result.textContent = `Sum: ${sum}`;
  });
</script>
```

Clicking the button displays the sum in the paragraph.

Questions and Answers on Classes vs IDs and Element Selection

Q8: What are the differences between Classes and IDs in HTML?

A8:

- **Classes:** Group multiple elements, can be shared, and an element can have multiple classes (e.g., `<p class="highlight">`).
- **IDs:** Uniquely identify one element, must be unique per page (e.g., `<div id="example">`).
- **Uniqueness:** Classes can repeat; IDs cannot.
- **Application:** Classes style multiple elements; IDs target specific elements for styling or JavaScript.

Q9: What does `querySelector()` do, and how is it used?

A9: `querySelector()` selects the first element matching a CSS selector. Syntax: `document.querySelector('selector')`. Example:

```
<div id="example">This is an example.</div>
<script>
  const element = document.querySelector('#example');
  element.style.color = 'blue';
</script>
```

It changes the text color of the element with ID "example" to blue.

Q10: How does `getElementById()` differ from `getElementsByClassName()`?
A10:

- **getElementById():** Selects one element by its unique ID (e.g., `document.getElementById('example')`).
- **getElementsByClassName():** Selects all elements with a specific class, returning a collection (e.g., `document.getElementsByClassName('highlight')`). Example:

```
<div id="example">Example</div>
<p class="highlight">Highlight</p>
<script>
  document.getElementById('example').style.color = 'red';
  const elements = document.getElementsByClassName('highlight');
  for (const el of elements) el.style.fontWeight = 'bold';
</script>
```

Questions and Answers on Communicating to the Backend

Q11: How does the frontend communicate with a backend server using `fetch`?
A11: The frontend uses the `fetch` API to send requests to a backend URL with parameters. Example:

```
<input id="firstNumber" type="text">
<input id="secondNumber" type="text">
<button onclick="populateDiv()">Calculate sum</button>
<div id="finalSum"></div>
<script>
  function populateDiv() {
    const a = document.getElementById("firstNumber").value;
    const b = document.getElementById("secondNumber").value;
    fetch("https://sum-server.100xdevs.com/sum?a=" + a + "&b=" + b)
      .then(response => response.text())
      .then(ans => document.getElementById("finalSum").innerHTML = ans);
  }
</script>
```

It fetches the sum from the backend and displays it in the `finalSum` div.

Q12: How does `async/await` improve backend communication compared to `.then()`?
A12: `async/await` makes the code more concise and readable by pausing execution until the Promise resolves. Example:

```
<script>
  async function populateDiv2() {
    const a = document.getElementById("firstNumber").value;
    const b = document.getElementById("secondNumber").value;
    const response = await fetch("https://sum-server.100xdevs.com/sum?a=" + a + "&b=" + b);
    const ans = await response.text();
    document.getElementById("finalSum").innerHTML = ans;
  }
</script>
```

It performs the same task as `populateDiv()` but with cleaner syntax.

Questions and Answers on onInput and Debouncing

Q13: What is the `onInput` event, and when is it triggered?
A13: The `onInput` event is an event handler triggered dynamically whenever a user changes an input field's value by typing or modifying it. It's used for real-time actions as the user interacts with the input.

Q14: What is debouncing, and why is it useful with `onInput`?
A14: Debouncing delays the execution of time-consuming tasks (e.g., requests) until the user stops typing for a specified duration. It's useful with `onInput` to reduce unnecessary calls, improving efficiency. Example:

```
<input id="textInput" onInput="debounce(handleInput, 500)">
<p id="displayText"></p>
<script>
  function debounce(func, delay) {
    let timeoutId;
    return function() {
      clearTimeout(timeoutId);
      timeoutId = setTimeout(() => func.apply(this, arguments), delay);
    };
  }
  function handleInput() {
    const value = document.getElementById("textInput").value;
    document.getElementById("displayText").innerText = "You typed: " + value;
    console.log("Request sent:", value);
  }
</script>
```

It waits 500ms after typing stops to update the display.

Q15: How does the `debounce` function work in the example?
A15: The `debounce` function takes a function (`func`) and delay, returning a new function that:

- Clears any existing timeout with `clearTimeout`.
- Sets a new timeout with `setTimeout` to call `func` after the delay.
  If the user keeps typing within 500ms, the timeout resets, ensuring `handleInput` runs only after a pause.

Questions and Answers on Throttling vs Rate Limiting

Q16: What is throttling, and what is its purpose?
A16: Throttling controls the rate at which an action is performed, limiting its frequency within a time frame. Its purpose is to ensure a smooth user experience by preventing rapid consecutive actions.

Q17: What is rate limiting, and how does it differ from throttling?
A17: Rate limiting controls the number of requests a client can make in a time period, protecting server resources and ensuring fairness. It differs from throttling, which focuses on action frequency for user experience, while rate limiting is server-side and limits request counts.

Q18: What are the key differences between throttling and rate limiting?
A18:

- **Focus:** Throttling controls action frequency; rate limiting controls request count.
- **Application:** Throttling applies to various actions; rate limiting is often API-level.
- **Goal:** Throttling enhances user experience; rate limiting protects server resources and enforces fairness.

Verification of Coverage

All critical topics from **Week 4.1** are addressed:

- **ECMA Scripts:** Defined and linked to JavaScript (Q1).
- **Auxiliary APIs:** Explained with examples (Q2-Q3).
- **DOM Manipulation:** Covered `document`, manipulation, and rendering (Q4-Q7).
- **Classes vs IDs:** Detailed differences (Q8).
- **Element Selection:** Explained `querySelector`, `getElementById`, and `getElementsByClassName` (Q9-Q10).
- **Backend Interaction:** Demonstrated with `fetch` and `async/await` (Q11-Q12).
- **onInput and Debouncing:** Defined and implemented (Q13-Q15).
- **Throttling vs Rate Limiting:** Compared key aspects (Q16-Q18).

Below is a set of questions and answers derived directly from the content you provided for "Week 8.3: Axios vs Fetch (Offline)." These focus on the essentials of the `fetch()` method, Axios, their differences, and practical code implementations, ensuring a comprehensive understanding of the material.

---

fetch() Method

Q1: What is the `fetch()` method in JavaScript, and what does it return?
A1: The `fetch()` method is a built-in JavaScript function that simplifies making HTTP requests to retrieve data from a server. It returns a Promise that resolves to the `Response` object representing the response to that request, whether successful or not.

Q2: Why is the `fetch()` method used in web development?
A2: It is used for:

1. **Asynchronous Data Retrieval**: It retrieves data asynchronously, allowing the code to continue without waiting, which is crucial for responsive web applications.
2. **Web API Interaction**: It simplifies making HTTP requests to external APIs.
3. **Promise-Based**: It returns a Promise, enabling clean asynchronous handling with `.then()` and `.catch()`.
4. **Flexible and Powerful**: It offers more flexibility and power than older methods like `XMLHttpRequest`, supporting various options and response types.

Q3: Provide a basic example of using the `fetch()` method from the content.
A3: Example:

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json();
  })
  .then(data => {
    console.log('Data from server:', data);
  })
  .catch(error => {
    console.error('Fetch error:', error);
  });
```

---

fetch() vs Axios

Q4: What are the key characteristics of the Fetch API?
A4: The key characteristics are:

1. **Native Browser API**: Built into modern browsers for HTTP requests.
2. **Promise-Based**: Returns a Promise, supporting modern asynchronous coding with `.then()` or `async/await`.
3. **Lightweight**: Bundled with browsers, requiring no external dependencies.

Q5: What are the key features of Axios?
A5: The key features are:

1. **External Library**: A standalone library for browsers and Node.js.
2. **Promise-Based**: Returns a Promise for consistent asynchronous handling.
3. **HTTP Request and Response Interceptors**: Allows global modification of requests or responses.
4. **Automatic JSON Parsing**: Automatically parses JSON responses.

Q6: How does `fetch` differ from Axios in terms of JSON parsing?
A6: With `fetch`, you need to manually call `.json()` on the response to parse JSON data. Axios automatically parses JSON responses, making it simpler as you can directly access

`response.data`.

Q7: What is a major advantage of Axios over `fetch` regarding request configuration?
A7: Axios allows detailed configuration of requests through a variety of options, whereas `fetch` requires more manual setup for headers, methods, and other configurations.

Q8: When should you use `fetch` instead of Axios?
A8: Use `fetch` when:

- Working on a modern project without needing additional features.
- Preferring a lightweight solution without external dependencies or polyfill concerns.

Q9: When should you choose Axios over `fetch`?
A9: Use Axios when:

- Dealing with complex scenarios requiring interceptors.
- Needing consistent behavior across different browsers.
- Desiring built-in features like automatic JSON parsing.

Comparison Points

Q10: How do `fetch` and Axios handle HTTP errors differently?
A10: Both allow error handling with `.catch()` or `.finally()`, but Axios may provide more detailed error information by default, while `fetch` requires checking `response.ok` and throwing errors manually for non-200 status codes.

Q11: What is a limitation of `fetch` compared to Axios in terms of interceptors?
A11: `fetch` lacks built-in support for interceptors, whereas Axios provides interceptors for globally modifying requests and responses before they reach `.then()` or `.catch()`.

Q12: How does browser support differ between `fetch` and Axios?
A12: `fetch` is natively supported in modern browsers but may need a polyfill for older browsers. Axios offers consistent behavior across various browsers and doesn’t rely on native implementations.

Q13: Which is lighter in terms of project size, `fetch` or Axios?
A13: `fetch` is lighter as it’s part of the browser, while Axios, being an external library, adds

additional file size to the project.

Comparing Code Implementation

Q14: How do you make a GET request using `fetch` with `async/await` according to the content?
A14: Example:

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
fetchData();
```

Q15: How do you make a GET request using Axios with `async/await`?
A15: Example:

```
import axios from 'axios';

async function fetchData() {
  try {
    const response = await axios.get('https://api.example.com/data');
    console.log(response.data);
  } catch (error) {
    console.error('Error:', error);
  }
}
fetchData();
```

Q16: Provide an example of a POST request using `fetch` with `async/await` from the content.

A16: Example:

```
async function postData() {
  const url = 'https://api.example.com/postData';
  const dataToSend = {
    key1: 'value1',
    key2: 'value2',
  };

  try {
    const response = await fetch(url, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(dataToSend),
    });
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
postData();
```

Q17: Provide an example of a POST request using Axios with `async/await` from the content.

A17: Example:

```
import axios from 'axios';

async function postData() {
  const url = 'https://api.example.com/postData';
  const dataToSend = {
    key1: 'value1',
    key2: 'value2',
  };

  try {
    const response = await axios.post(url, dataToSend);
    console.log(response.data);
  } catch (error) {
    console.error('Error:', error);
  }
}
postData();
```

Q18: What is a key difference in sending data between GET and POST requests as noted in the content?
A18: In GET requests, you can send headers but not a body, while in POST, DELETE, and PUT requests, you can send both headers and a body.

These questions and answers cover the core concepts from "Week 8.3: Axios vs Fetch (Offline)," including the functionality of `fetch()`, the features of Axios, their differences, and practical code examples. They ensure a solid grasp of data fetching techniques in web development as presented in the material.