Advanced Routing

• Setting up dynamic routes and route parameters

• Using query strings and request body parsing

• Organizing routes using Express Router

• Implementing route-specific middleware

Dynamic Route Is

- **Concept**: A dynamic route is a URL path that can change based on the information we need to pass. For example, if we have a user profile page, each user has a unique ID. Instead of creating a separate route for each user, we use a dynamic route that adapts based on which user we're looking up.

Basic Syntax for Dynamic Routes

- Explain that in Express.js, we define a dynamic part of a route by adding a colon : before the parameter name. For example:

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id; // This captures the value of the ID in the URL
  res.send(`User ID is: ${userId}`);
});
```

Here, :id is a placeholder that can be any value. When we access /users/123, Express will treat 123 as id and make it available in req.params.id.

Multiple Route Parameters

- You can add more than one dynamic parameter if needed. For example, /users/:userId/posts/:postId can be used to see specific posts by a user.

```
app.get('/users/:userId/posts/:postId', (req, res) => {
  const { userId, postId } = req.params;
  res.send(`User ID: ${userId}, Post ID: ${postId}`);
});
```

Benefits of Dynamic Routes

- **Flexibility**: You don't need multiple routes for each user, product, or post.
- **Maintainability**: Code is cleaner and easier to manage, especially in large applications.

Query Strings

- **What are Query Strings?** Query strings are parameters sent as part of the URL after a question mark (?). They're often used to filter, search, or pass extra information in a request.
- **Structure of Query Strings**: Query strings are typically written as key=value pairs and can be combined using the ampersand (&). For example:

```
http://example.com/search?term=javascript&sort=asc
```

In this URL, term=javascript and sort=asc are query strings.

Accessing Query Strings in Express: Express makes it easy to access query strings using req.query, which returns an object containing all query parameters.

Example Code: Here's how to create a route that captures a query string for a search term

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

app.get('/search', (req, res) => {
  const searchTerm = req.query.term; // Accessing the 'term' query string
  const sortBy = req.query.sort; // Accessing the 'sort' query string

  res.send(`Search term is: ${searchTerm}, sorted by: ${sortBy}`);
});

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

In this example, if a user visits http://localhost:3000/search?term=javascript&sort=asc, the server captures term=javascript and sort=asc from req.query and sends them in the response.

- **Practical Use Case**: Query strings are commonly used for filtering results, such as showing a list of products with specific attributes (e.g., category, price range) or sorting order.

Express Router

- **Modularity**: Separates different parts of your application, such as user-related and product-related routes, into separate files.
- **Maintainability**: Keeping routes organized makes code easier to read, debug, and extend.
- **Scalability**: Allows you to add new routes or route groups without cluttering the main app file.

Setting Up Express Router

Let's say you have a basic API with user-related and product-related routes.

1. **Create the Project Structure**

```
my-app/
├── routes/
│   ├── users.js
│   └── products.js
├── app.js
└── package.json
```

app.js will be the main file where the app is initialized.
routes/users.js and **routes/products.js** will contain the code specific to users and products, respectively.

Create Route Files

- **routes/users.js**

```
const express = require('express');
const router = express.Router();

// Define a route to get user details
router.get('/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID is: ${userId}`);
});

// Define a route to create a new user
router.post('/', (req, res) => {
  res.send('User created successfully');
});

module.exports = router;
```

- Here, we're creating two routes in users.js: one for getting user details by ID and another for creating a new user.
- We export router so it can be used in the main app.js file.

routes/products.js

```
const express = require('express');
const router = express.Router();
```

```
// Define a route to get product details
router.get('/:id', (req, res) => {
  const productId = req.params.id;
  res.send(`Product ID is: ${productId}`);
});

// Define a route to create a new product
router.post('/', (req, res) => {
  res.send('Product created successfully');
});

module.exports = router;
```

Similarly, in products.js, we define routes to get product details and create a product.

Integrate Routers in app.js

- **app.js**

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

// Import the routes
const userRoutes = require('./routes/users');
const productRoutes = require('./routes/products');

// Middleware to parse JSON bodies
app.use(express.json());

// Use routes with base paths
app.use('/users', userRoutes);      // All routes in users.js are prefixed with /users
app.use('/products', productRoutes); // All routes in products.js are prefixed with /products

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

Here, app.use('/users', userRoutes) tells Express that all routes in users.js should start with /users.

- For example, a GET request to /users/123 will trigger the route in users.js for fetching user details by ID.

Similarly, app.use('/products', productRoutes) makes all routes in products.js start with /products.

Testing the Routes

- Run the app with node app.js.
- Test the following routes:
    - **GET** http://localhost:3000/users/1 – Should respond with User ID is: 1.
    - **POST** http://localhost:3000/users – Should respond with User created successfully.
    - **GET** http://localhost:3000/products/10 – Should respond with Product ID is: 10.
    - **POST** http://localhost:3000/products – Should respond with Product created successfully.

Implementing route-specific middleware

What is Middleware?

- Middleware in Express is a function that has access to the req, res, and next objects.
- It processes requests and responses, and it can:
    - Modify the request object.
    - End the request-response cycle.
    - Call next() to pass control to the next middleware function.

Why Use Route-Specific Middleware?

- Instead of applying a middleware function globally (for all routes), route-specific middleware allows you to control where the middleware runs.
- This is useful when you only need specific checks or transformations on certain routes.

Creating a Route-Specific Middleware

- Let's say you have a route for getting user details, but you want to make sure that only logged-in users can access this route.
- **Example Middleware Function**:

```
// Middleware to check for valid authorization in query parameter
function checkAuthQuery(req, res, next) {
  const authKey = req.query.auth; // Check the 'auth' query parameter

  // Simple check: Allow access if auth key matches "mysecurekey"
  if (authKey === 'mysecurekey') {
    next(); // Auth key is valid, proceed to route handler
  } else {
    res.status(403).send('Access Forbidden: Invalid Authentication Key'); // Invalid auth key, respond with 403 Forbidden
  }
}
```

Applying Route-Specific Middleware

- You can apply this middleware to individual routes where authentication is needed. Let's apply it to a route that gets user details.

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

// Sample data
const orders = [
  { id: 1, item: 'Laptop', quantity: 1 },
  { id: 2, item: 'Phone', quantity: 2 },
  { id: 3, item: 'Tablet', quantity: 3 },
];

// Middleware to check for valid authorization in query parameter
function checkAuthQuery(req, res, next) {
  const authKey = req.query.auth; // Check the 'auth' query parameter

  // Simple check: Allow access if auth key matches "mysecurekey"
  if (authKey === 'mysecurekey') {
    next(); // Auth key is valid, proceed to route handler
  } else {
    res.status(403).send('Access Forbidden: Invalid Authentication Key'); // Invalid auth key,
respond with 403 Forbidden
  }
}

// Protected route to get orders
app.get('/orders', checkAuthQuery, (req, res) => {
  res.json(orders); // Send the list of orders if authorized
});

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

Explanation

- **Middleware**: checkAuthQuery checks if the auth query parameter matches a predefined key (mysecurekey).
- **Protected Route**: The /orders route uses checkAuthQuery to allow access only if the auth query parameter is valid.

Testing

- Access http://localhost:3000/orders?auth=mysecurekey to view the orders.
- If the auth key is missing or incorrect, the server will respond with a 403 Forbidden error.

Interview questions:

What are dynamic routes in Express.js, and how do they work?

Answer:
Dynamic routes in Express.js allow us to define routes that contain variables or parameters in the URL. These parameters are denoted by a colon (:) followed by the parameter name (e.g., /users/:id). When a request is made to a dynamic route, Express extracts the parameter values from the URL and makes them available in req.params.

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID is: ${userId}`);
});
```

In this example, if a user accesses /users/123, req.params.id would be 123.

How can we use multiple route parameters in a single route?

Answer:
In Express.js, you can define multiple route parameters by adding more :parameterName segments to the route path. For example, /users/:userId/posts/:postId allows access to both userId and postId parameters via req.params.userId and req.params.postId.

```
app.get('/users/:userId/posts/:postId', (req, res) => {
  const { userId, postId } = req.params;
  res.send(`User ID: ${userId}, Post ID: ${postId}`);
});
```

What is the difference between req.params, req.query, and req.body?

Answer:

- req.params: Contains route parameters defined in the URL path (e.g., /users/:id).
- req.query: Contains query string parameters, which are appended to the URL after the ? symbol (e.g., /search?term=javascript).
- req.body: Contains data sent in the request body, typically used in POST or PUT requests. To access req.body, Express needs middleware (express.json() for JSON data or express.urlencoded() for form data).

How do you access query parameters in Express.js?

Answer:
In Express, query parameters are accessed via req.query, which is an object containing all query string parameters as key-value pairs. For example, if the URL is /search?term=javascript, then req.query.term would be "javascript"

```
app.get('/search', (req, res) => {
```

```
  const searchTerm = req.query.term;
  res.send(`Search term is: ${searchTerm}`);
});
```

Why is it necessary to use middleware like express.json() or express.urlencoded() in Express?

Answer:
Express doesn't parse request bodies by default. The express.json() middleware is required to parse incoming JSON data in the request body, and express.urlencoded() is needed for parsing URL-encoded form data. This enables access to the parsed data via req.body.

What is Express Router, and why is it useful?

Answer:
Express Router is a built-in feature that allows you to organize routes into separate, modular files or groups. This is especially useful for larger applications, as it helps keep the code organized, maintainable, and scalable. Each instance of express.Router() acts as a mini-application that can define its own routes and middlewares.

How do you organize routes into separate files using Express Router?

Answer:
To organize routes, create a new file for each route group, use express.Router() to define routes, and export the router. Then, import and mount the router in the main application file using app.use(). For example, to create user-related routes:

1. In routes/users.js:

```
const express = require('express');
const router = express.Router();

router.get('/:id', (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});

module.exports = router;
```

In app.js:

```
const express = require('express');
const app = express();
const userRoutes = require('./routes/users');

app.use('/users', userRoutes);
```

This makes all routes in userRoutes available under the /users path.

What are the benefits of organizing routes using Express Router?

Answer:
Organizing routes with Express Router improves code readability and maintainability, especially in large applications. It allows for modular development, enabling teams to work on different route groups independently. It also makes it easier to add middlewares and features without cluttering the main application file.

What is route-specific middleware, and how is it different from application-wide middleware?

Answer:
Route-specific middleware applies only to certain routes, whereas application-wide middleware applies to all incoming requests. In Express, you can add middleware functions as arguments to specific routes. This approach is useful for tasks like authentication, validation, or logging that should only affect certain endpoints.

```
function checkAuthentication(req, res, next) {
  if (req.query.auth === 'validkey') {
    next(); // Authenticated, proceed
  } else {
    res.status(401).send('Unauthorized');
  }
}

app.get('/profile', checkAuthentication, (req, res) => {
  res.send('User profile data');
});
```

Can you add multiple middleware functions to a single route? How?

Answer:
Yes, you can chain multiple middleware functions on a single route by listing them as arguments. Each middleware function should call next() to pass control to the next function.

```
function logRequest(req, res, next) {
  console.log(`Request URL: ${req.url}`);
  next();
}

function checkAuthentication(req, res, next) {
  if (req.query.auth === 'validkey') {
    next();
  } else {
    res.status(401).send('Unauthorized');
  }
}

app.get('/profile', logRequest, checkAuthentication, (req, res) => {
  res.send('User profile data');
});
```