# EXPRESS STUDY NOTES

## 1. Introduction to Express.js

**Explanation:** Express.js is a web framework built on top of **Node.js**. It simplifies the process of routing, handling HTTP requests, and rendering dynamic content. Express is lightweight and modular, allowing you to build both APIs and full-stack applications.

**Code Example:**

javascript
Copy code

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

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

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

---

## 2. Setting Up Express.js

**Explanation:** To start using Express, you first need to install it using **npm**. You can then create an Express app and set up routing to handle requests.

**Installation:**

bash
Copy code

```
npm init -y
npm install express
```

**Code Example:**

javascript
Copy code

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

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

// A basic route that responds with "Hello, Express!"
app.get('/', (req, res) => {
  res.send('Hello, Express!');
});

// Start the server
app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

---

### 3. Routing in Express

**Explanation:** Routing refers to how the application responds to client requests. You can define different routes for different HTTP methods like `GET`, `POST`, `PUT`, and `DELETE`.

**Code Example:**

javascript
Copy code

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

// Respond to GET request at "/"
app.get('/', (req, res) => {
  res.send('GET Request to Home Page');
});

// Respond to POST request at "/login"
app.post('/login', (req, res) => {
  res.send('POST Request to Login');
});

// Respond to PUT request at "/update"
app.put('/update', (req, res) => {
  res.send('PUT Request to Update Information');
});

// Respond to DELETE request at "/delete"
```

```
app.delete('/delete', (req, res) => {
  res.send('DELETE Request to Remove Data');
});

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

---

### 4. Middleware in Express

**Explanation:** Middleware is a function that is executed during the request-response cycle. It has access to the request object, the response object, and a `next` function that passes control to the next middleware or route handler.

**Code Example:**

javascript
Copy code

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

// Custom middleware to log the request time
app.use((req, res, next) => {
  console.log('Request received at:', new Date());
  next(); // Pass control to the next handler
});

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

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

**Explanation of middleware:**

- `app.use()` is used to add middleware that will be executed for all incoming requests.
- Middleware functions can modify request or response objects or perform other tasks before passing control to the next middleware.

---

## 5. Handling Requests and Responses

**Explanation:** Express simplifies handling different types of HTTP requests (e.g., GET, POST) and sending different types of responses (e.g., JSON, text, HTML).

**Code Example:**

javascript
Copy code

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

// Handling GET request and sending a text response
app.get('/text', (req, res) => {
  res.send('This is a text response');
});

// Handling GET request and sending a JSON response
app.get('/json', (req, res) => {
  res.json({ message: 'This is a JSON response' });
});

// Handling GET request and sending an HTML response
app.get('/html', (req, res) => {
  res.send('<h1>Welcome to Express!</h1>');
});

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

---

## 6. Express Template Engines

**Explanation:** Express supports template engines like **Pug**, **EJS**, and **Handlebars** for rendering dynamic HTML pages. You can pass dynamic data from your routes to these templates, and they will generate HTML on the server-side.

**Code Example (using Pug):**

**Install Pug:**
bash
Copy code
```
npm install pug
```

1.
2. **Set Pug as the template engine:**

javascript
Copy code
```
const express = require('express');
const app = express();

// Set Pug as the view engine
app.set('view engine', 'pug');

// Render a Pug template with dynamic data
app.get('/home', (req, res) => {
  res.render('home', { name: 'John Doe' });
});

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

3. **Create a `views/home.pug` file:**

pug
Copy code
```
h1 Welcome #{name}
```

---

## 7. Handling Forms in Express

**Explanation:** Express can handle form submissions using `POST` requests. You'll need to parse the form data using `express.urlencoded()` middleware.

**Code Example:**

javascript
Copy code

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

// Middleware to parse URL-encoded data (from forms)
app.use(express.urlencoded({ extended: true }));

// Handling a form submission (POST request)
app.post('/submit', (req, res) => {
  const { username, email } = req.body;
  res.send(`Received submission: Username - ${username}, Email -
${email}`);
});

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

**Explanation:**

- `express.urlencoded({ extended: true })` middleware is used to parse form data (sent via POST).
- The form data is accessible via `req.body`.

---

## 8. Connecting to Databases in Express

**Explanation:** Express can be connected to databases like **MongoDB** or **MySQL** to store or retrieve data.

**Code Example (using MongoDB with Mongoose):**

**Install Mongoose:**
bash
Copy code

```
npm install mongoose
```

1.
2. **Connect to MongoDB and define a model:**

javascript
Copy code
```
const mongoose = require('mongoose');

// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/myapp', {
useNewUrlParser: true, useUnifiedTopology: true });

// Define a User schema
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
});

// Create a User model
const User = mongoose.model('User', userSchema);

// Create a new user and save it to the database
app.post('/create', async (req, res) => {
  const user = new User({ name: 'John Doe', email:
'john@example.com' });
  await user.save();
  res.send('User saved');
});

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

---

## 9. Authentication and Security in Express

**Explanation:** Authentication and security are essential parts of any web application. Express can handle user authentication using libraries like **Passport.js** and implement security features using **Helmet.js**.

**Code Example (using Passport.js for authentication):**

**Install Passport.js:**
bash
Copy code
```
npm install passport passport-local express-session
```

1.
2. **Configure Passport for user authentication:**

javascript
Copy code

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

// Use LocalStrategy for authentication
passport.use(new LocalStrategy((username, password, done) => {
  // In a real app, check the database for user
  if (username === 'admin' && password === 'password') {
    return done(null, { username });
  } else {
    return done(null, false, { message: 'Invalid credentials' });
  }
}));

// Session setup
passport.serializeUser((user, done) => {
  done(null, user.username);
});

passport.deserializeUser((username, done) => {
  done(null, { username });
});

// Initialize Passport and session handling
app.use(express.session({ secret: 'secret-key' }));
app.use(passport.initialize());
app.use(passport.session());

// Login route
app.post('/login', passport.authenticate('local', {
  successRedirect: '/profile',
  failureRedirect: '/login',
}));
```

---

## 10. Error Handling in Express

**Explanation:** Error handling ensures that when something goes wrong, the application can respond gracefully and give users appropriate feedback.

**Code Example:**

javascript
Copy code

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

// Custom error-handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something went wrong!');
});

// Route to trigger an error
app.get('/error', (req, res) => {
  throw new Error('Something went wrong!');
});

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

**Explanation of Error Handling:**

- Express allows defining custom error-handling middleware after all routes.
- If an error occurs in any route, it passes the error to the next middleware function, which is defined for error handling.

---

## Conclusion

Express.js simplifies web application development with its routing, middleware, templating, and error handling features. By integrating databases, authentication, and security measures, you can build robust and scalable applications.

Let me know if you need any further clarification or specific examples!