Express.js

Node.js Express

This module assumes that students understand the following:

1. ASE 220 Totpics: MongoDB

- We discuss MongoDB and Mongoose in `mongodb_and_mongoose`

3. ASE 230 Topics: REST API & Server Side Application Development

- PHP/Laravel and JavaScript/AJAX

What Node.js Is

- JavaScript runtime (V8 + libuv)
- Your **server runs inside Node**, not Apache/Nginx
- Persistent process instead of “run PHP per request”

PHP model:

Request → PHP runs → exits

Node model:

Server always running → handles many requests

What Express Is (Compared to PHP/Laravel)

- Minimal web framework for Node.js
- Similar to **Laravel routing layer**
- Provides:
  - Routing
  - Middleware
  - Request/response helpers
- Does **not** include:
  - ORM
  - Templating system (optional)
  - Auth scaffolding

Routing Comparison

PHP (Laravel)

```php
Route::get('/posts', function () {
    return Post::all();
});
```

Node.js (Express)

```javascript
app.get('/posts', async (req, res) => {
  const posts = await postsCol.find().toArray();
  res.json(posts);
});
```

Same REST concept, different environment.

Request Model

PHP

- Web server invokes PHP per request
- Clean fresh state on each request

Node.js

- Express server stays running
- Can keep:
  - DB connections
  - Caches
  - Variables in memory

Great for:

- WebSockets (We will use WebSocket in the project)
- Streaming
- Live apps

Middleware

Middleware = functions that run before your route handler.

Equivalent to Laravel middleware, but simple to use without `php artisan make:middleware`.

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

Laravel → Middleware + Service Providers + Filters

Express → Everything is middleware

Views (EJS)

Using EJS (like Blade):

```
app.get('/', (req, res) => {
  res.render('home', { posts: posts });
});
```

This is EJS Template (home.ejs).

```
<ul class="list-group">
<% for (var i = 0; i < posts.length; i ++) { %>
    <h4><%= posts[i].title %></h4>
    <p><%= posts[i].date %></p>
<% } %>
</ul>
```

Folder Structure (Laravel-Like)

```
/routes
/controllers
/models
/views
```

Express does NOT enforce structure, but this is common.

Express Example (index_express.js)

We use MongoDB as the backend Database.

```
const { getDB } = require('./util/db');

async function initialize() {
  db = await getDB(DATABASE);
}
initialize();
```

- In this example, when we use CommonJS module, we cannot use `await` at the top level, so we should put the async/await in the file.
- Refer to `mongodb_and_mongoose` for using MongoDB.

Express application

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

- The express is the Express framework function.
- Calling express() creates an application object.

app = the Express application used to process REST API requests and send responses.

Using Middleware

This middleware allows Express to parse data from HTML forms (sent using POST, PUT, PATCH, or DELETE).

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

It parses requests with : `Content-Type: application/x-www-form-urlencoded`

This is the default format used by:

- <form method="POST"> in HTML
- Standard browser form submissions
- Simple API clients

After using it, Express places the parsed values into: `req.body`

Body-parser

Without body-parser, Express could not read `req.body`.

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

- Parsing JSON request bodies
- Parsing URL-encoded form data (e.g., HTML <form> submissions)
- Putting parsed data into req.body

Callback functions

In this example, the server processes GET request to '/'.

```
app.get('/', function (req, resp) {
  resp.sendFile(__dirname + '/write.html')
});
```

- When the client goes to http://localhost:5500/
- Express runs this handler
- It sends the file write.html back to the browser
- This is usually a form page that submits data

The server processes POST request of add API.

```
app.post('/add', async function (req, resp) {
  resp.send('Sent');
  try {
    const posts = db.collection(`${COLLECTION}`);
    const query = { title: req.body.title, date: req.body.date }
    await posts.insertOne(query);
  } catch (e) {
    console.error(e);
  }
});
```

1. Waits for a POST submission
2. Reads form data from req.body
3. Immediately sends "Sent" back to the client
4. Inserts the data into MongoDB

The Request method and endpoint

The function of Express app matches the request method and the first argument matches the endpoint (API).

```
app.get('/', function); // GET method / root
app.post('/add', function); // POST method /add API
app.delete('/delete', function); // DELETE method /delete API
```

runAddPost function

We need to separate the routing and business logic (algorithm) to make the code read and maintain to easy.

```
app.post('/add', function(req, resp) {
    runAddPost(req, resp);
});

async function runAddPost(req, resp) {
  try {
    ...
  }
```

In this example, we use `counter` collection to keep track of the ID.

The JSON representation of the file is as follows:

```
{
  name: "count"
  count: N
}
```

There is only one file in this collection; we can access this file using name = "count", and the total count (N) is accessed using count variable.

```
async function runAddPost(req, resp) {
  try {
    const counter = db.collection(COUNTER);
    const result = await counter.findOneAndUpdate(
      { name: 'count' },
      { $inc: { count: 1 } },
      { returnDocument: 'after', upsert: true }
    );
    ...
  }
```

1. `$inc: { count: 1 }`: Increments the `count` field by 1 (creates it if missing).
2. `returnDocument: 'after'`: Returns the document after the update.
3. `upsert: true`: Inserts a new document if none matches the filter.

Using the new ID (newId), we create a title and date from `req.body` to create one post in the POST collection.

```
    const newId = result.count; // now safe to access
    const newPost = {
      _id: newId,
      title: req.body.title,
      date: req.body.date
    };
    await posts.insertOne(newPost);

    resp.send('Stored to MongoDB OK');

  } catch (e) {
    console.error(e);
  }
```

We return a string 'Stored to MongoDB OK' to the client.

Start the Server

```
app.listen(5500, function () {
  console.log('listening on 5500')
});
```

- The client can access the server from http://localhost:5500/.
- We can change the port number by replacing 5500 with any port number.