EXPRESS.JS

Routes & REST

WHAT'S ON THE SCHEDULE?

- Pop Quiz
  - :) :) :)
- Express Router
- REST pattern
- Body Parser

POP QUIZ

EXPRESS ROUTER

EXPRESS ROUTER

- Express provides a Router middleware to create modular, mountable route handlers.
- Think of it as a “mini-app” that nests within an existing app.
- It lets you break up the major parts of your application into separate modules.

App.js

```
const express = require("express");
const morgan = require("morgan");
const client = require("./db");
const postList = require("./views/postList");
const postDetails = require("./views/postDetails");

const app = express();

app.use(morgan("dev"));
app.use(express.static(__dirname + "/public"));

app.get("/", async (req, res) => {
  const data = await client.query("SELECT...");
  res.send(postList(data.rows));
});

app.get("/posts/:id", async (req, res) => {
  const data = await client.query("SELECT ...);
  const post = data.rows[0];
  res.send(postDetails(post));
});

const PORT = 1337;

app.listen(PORT, () => {
  console.log(`App listening in port ${PORT}`);
});
```

App.js

```javascript
const express = require("express");
const morgan = require("morgan");


const routes = require("./routes");

const app = express();

app.use(morgan("dev"));
app.use(express.static(__dirname + "/public"));
app.use(routes);

const PORT = 1337;

app.listen(PORT, () => {
  console.log(`App listening in port ${PORT}`);
});
```

routes.js

```javascript
const express = require('express');
const router = express.Router();
const client = require("./db");
const postList = require("./views/postList");
const postDetails = require("./views/postDetails");

app.get("/", async (req, res) => {
  const data = await client.query("SELECT...");
  res.send(postList(data.rows));
});

app.get("/posts/:id", async (req, res) => {
  const data = await client.query("SELECT ...);
  const post = data.rows[0];
  res.send(postDetails(post));
});

module.exports = router;
```

App.js

```
const express = require("express");
const morgan = require("morgan");


const routes = require("./routes");

const app = express();

app.use(morgan("dev"));
app.use(express.static(__dirname + "/public"));
app.use(routes);

const PORT = 1337;

app.listen(PORT, () => {
  console.log(`App listening in port ${PORT}`);
});
```

routes.js

```
const express = require('express');
const router = express.Router();
const client = require("./db");
const postList = require("./views/postList");
const postDetails = require("./views/postDetails");

router.get("/", async (req, res) => {
  const data = await client.query("SELECT...");
  res.send(postList(data.rows));
});

router.get(“/posts/:id", async (req, res) => {
  const data = await client.query("SELECT ...);
  const post = data.rows[0];
  res.send(postDetails(post));
});

module.exports = router;
```

App.js

```
const express = require("express");
const app = express();
app.use(morgan("dev"));
app.use(express.static(__dirname + "/public"));

app.use('/posts', require('./routes/posts'));
app.use('/users', require('./routes/users'));

const PORT = 1337;

app.listen(PORT, () => {
  console.log(`App listening in port ${PORT}`);
});
```

posts.js **users.js**

```
const express = require('express');
const router = express.Router();
const client = require("./db");

router.get("/", async (req, res) => {
  const data = await client.query("SELECT...");
  res.send(postList(data.rows));
});

router.get("/:id", async (req, res) => {
  const data = await client.query("SELECT ...);
  const post = data.rows[0];
  res.send(postDetails(post));
});

module.exports = router;
```

REST

REST

- Architecture style for designing backend applications.
- Helps answer the question on how to organize routes and how to map functionality to URIs and Methods:
  - Paths represent "nouns" or resources
  - HTTP “verbs” map to data operations

REST - RESOURCES

GET /a rs

GET /user/

GET /ad e=“Rubeus”

REST - RESOURCES

GET	/users	Show all users
GET	/users/4	Show a single user (whose ID=4 in the db)
POST	/users	Create a new user in the DB
PUT	/users/4	Update user 4 in the db
DELETE	/users/4	Delete user 4 from the db

COMMON MISTAKES (NON REST-FUL)

1. MIXING VERBS & NOUNS

The feature

- We need to delete individual records from the bears table

THE MISCONCEPTION

```
❌ GET /bears/delete/:id
❌ GET /bears/:id?delete=true
```

These url attempts to specify the operation to take in the url path and query string. Paths should only convey information about the resource. The RESTful way to do this would be `DELETE /bears/1`, not `GET /bears/delete/1`. GET is also supposed to be "safe" (does not affect the backend in an observable way).

SOLUTION

✅ `DELETE /bears/:id`

2. OPAQUE RESOURCE

The feature

- It’s a historical chess database. We have pages for individual moves in historic games.

THE MISCONCEPTION

❌ `GET /games/:gameId/:moveNumber`

`GET /games/4/2` is not expressive enough. What does "2" refer to? The only way to know is by inspecting the implementation (or having documentation)—it's not self-evident. The RESTful way to do this is to explicitly include the sub-resource as a label: `GET /games/4/moves/2`. Modify the path.

SOLUTION

✅ `GET /games/:gameId/moves/:moveNumber`

3. MISLEADING RESOURCE

The feature

- We’re a scientific laboratory. Our employees generate lots of reports. We need to get a list of all the reports generated by particular request

THE MISCONCEPTION

```
❌ GET /reports/:scientistId
```

`GET /reports/4` means "get report #4", NOT "get all reports by scientist #4". There are two good solutions for this, and they are not mutually exclusive—you can use both.

SOLUTION 1: SUB-RESOURCE FOR USERS

✅ `GET /users/:scientistId/reports`

SOLUTION 2: QUERY STRING

✅ `GET /reports?scientistId=4`

4. UNPREDICTABLE URI STRUCTURE

The feature

- We're a big e-commerce site. We want to show all the reviews for a particular project on the same page

THE MISCONCEPTION

❌ `GET /reviews/products/:productId`

As a rule of thumb we expect RESTful URIs to follow the pattern `/foo/:fooID/bar/:barID/baz/:bazID` etc. REST is all about predictability via consistency, and `/reviews/products/4` contradicts that—`reviews` is not followed by an identifying key, it is followed by a *separate* resource name (`products`).

Another way to think about it is that it is similar to **Misleading Resource**. The `GET` request indicates that we're getting something, but what are we getting? One product? Many products? One review? Many reviews?

SOLUTION 1: SUB-RESOURCE FOR REVIEWS

✅ `GET /products/:productId/reviews`

SOLUTION 2: QUERY STRING

✅ `GET /reviews?productId=4`

REQUEST BODY & BODY PARSER

- POST & PUT HTTP requests can contain information in the body
- The request body is streamed and frequently compressed
- Express comes with a built-in middleware that automatically parses incoming request bodies and makes the data available under `req.body`

BODY PARSER

verb route

headers

```
POST /books HTTP/1.1
Host: www.test101.com
Accept: */*
Acce
Acce
User                                                   .1)

bookId=12345&author=Nimit
```

In express…

```
req.body = {bookId:12345, author: 'Nimit'}
```

body

BODY PARSER

```
const express = require('express');
app.use(express.urlencoded({ extended: false }));
```

WHY BODY PARSER?

How the Internet Works