

Express.js

What is Express.js?

- Minimalist framework
- Very similar to Flask
- Most popular in the Node ecosystem

A Server In A Few Lines

```
demo/tiny.js

const express = require('express');

const app = express();

app.listen(3000, function () {
  console.log('App on port 3000!');
})
```

- App doesn't do anything except respond 404s, but server is running!
- **app.listen** takes a port and a callback.
 - Binds the server to port & listens for requests there.
 - Calls callback once server has started up.
- **app.listen** should always be at the bottom of the file

How Does Express Work?

Route Handlers

Route handlers are event listeners – they're like Flask view functions

```
demo/firstRoute.js      demo/firstRoute.py

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

app.get('/dogs', function(request, response) {
  response.send('Dogs go brk brk!');
});

app.listen(3000, function(){
  console.log('App on port 3000!');
})

from flask import Flask
app = Flask(__name__)

@app.route('/dogs')
def bark():
    return 'Dogs go brk brk'
```

app.get('/dogs') listens for a **GET** Request to the **/dogs** endpoint.

In the callback, **response.send()** issues a response of plain-text or HTML.

Route Handler Callbacks

- Every handler should have a callback with two parameters:
 - **request**: information about request (query string, url parameters, form data)
 - **response**: useful methods for sending a **response** (html, text, json, etc.)
- You will commonly see these parameters named **req** and **res**

Express builds **req** and **res** objects for every request and passes them to callback.

The Request-Response Cycle

When you start the server, Express runs through the file and registers all the event listeners before **app.listen** at the bottom.

Whenever a user makes a request, Express invokes the **first matching route handler it finds** until a response is issued via a method on the **response** object.

This is called the **request-response cycle** for Express.

```
demo/secondRoute.js

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

app.get('/dogs', function(req, res) {
  return res.send('Dogs go brk brk!');
});

// this will never get matched
app.get('/dogs', function(req, res) {
  return res.send('but what about these dogs???');
});

app.listen(3000, function () {
  console.log('App on port 3000!');
})
```

First route handler gets matched because it was registered first.

Second handler never matched because a response is issued in the previous handler, thus concluding the request cycle.

Route Methods

Here are the route methods you will likely use, one for each HTTP verb:

- **app.get(path, callback)**
- **app.post(path, callback)**
- **app.put(path, callback)**
- **app.patch(path, callback)**
- **app.delete(path, callback)**

Our First Express App

Getting started

```
$ mkdir first-express-app
$ cd first-express-app
$ npm init -y
$ npm install express
```

Nodemon

- Nodemon restarts server when you edit files or if the server crashes.
- Install globally, so you can use in any project:

```
$ npm install --global nodemon

$ nodemon app.js
```

URL Parameters

Specify parameters in the route by prefixing them with a colon : .

```
demo/app.js

/* Show info on instructor. */

app.get('/staff/:fname', function(req, res) {
  return res.send('This instructor is ' + req.params.fname);
});

• All of our parameters become keys in an object found at req.params
• The values are always strings
```

Other Useful Request Properties

- query string (**request.query**)
- headers (**request.headers**)
- what about the body of the request?

Parsing the Body

Tell Express to parse request bodies for either form data or JSON:

```
demo/app.js

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

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

To access the parsed request body: req.body
```

```
demo/app.js

/* Add a new instructor. */

app.post('/api/staff', function(req, res) {
  // Do some database operation here...
  return res.send({
    fname: req.body.fname
  });
});
```

Warning: body-parser

A recent update to Express added the method **express.json()**. Previously you had to utilize a small library called **body-parser** to do this, so if you see body-parser, it does the same thing.

Returning JSON

It's dead simple!

```
demo/app.js

/* Show JSON on instructor */

app.get('/api/staff/:fname', function(req, res) {
  return res.json({ fname: req.params.fname });
});
```

Status Codes

To issue status codes with our responses, we can call the **res.status(code)** method first, and then chain our **json()** to finish the response.

```
demo/app.js

/* Sample of returning status code */

app.get('/whoops', function(req, res) {
  return res.status(404).json('Whoops! Nothing here!');
});
```

Validation and Errors

```
demo/app.js

/* Sample of validating / error handling */

app.get('/dogs:name', function(req, res) {
  if (req.params.name !== 'Whiskey') {
    return res
      .status(403)
      .json('Only Whiskey is Allowed.');
```

This will work to start, but we can do better!

Error Handling in Express

How do we let Express know about an error?

There are quite a few ways, but the easiest is just to **throw** an error!

```
app.get("/users/:username", function(req,res,next) {
  const user = USERS.find(u => u.username === req.params.username);

  if (!user) throw "Not found!";

  return res.send({user});
})
```

A couple issues here

- We can't easily see the stack trace
- What about attaching a status code like 404 or 401?
- If we want that kind of flexibility, we need to create it!

Let's make our own error class!

```
demo/routing-app/expressError.js

/* ExpressError extends the normal JS error so we can easily
 * add a status when we make an instance of it.
 *
 * The error-handling middleware will return this.
 */

class ExpressError extends Error {
  constructor(message, status) {
    super();
    this.message = message;
    this.status = status;
    console.error(this,stack);
  }
}

module.exports = ExpressError;
```

Throwing an error

Now that we have our own custom error class, let's use it!

```
const ExpressError = require("./expressError")

app.get("/users/:username", function(req, res) {
  const user = USERS.find(u => u.username === req.params.username);

  if (!user) throw new ExpressError("Not found!", 404);

  return res.send({user});
})
```

Almost there.....

- Now throwing errors, but server never responds with anything.
- We need to instruct Express how to respond when we throw errors.
- For that, we need to introduce two new concepts, error handling and **next**

Error Handling in express

In Express, error handlers are special types of handlers. Here are the rules for building an error handler:

- Error handlers should be at the bottom of the file, just above **app.listen**. This is because *any* handlers defined above can potentially throw errors!
- They should match every HTTP verb and path: **app.use(callback)**.
- Callback signature to error handlers has 4 parameters instead of 3
 - **function (error, req, res, next)**
 - (This is how Express knows it's an error-handler)

Global Error Handler Example

```
demo/routing-app/app.js

app.use(function(err, req, res, next) {
  // the default status is 500 Internal Server Error
  let status = err.status || 500;
  let message = err.message;

  // set the status and alert the user
  return res.status(status).json({
    error: {message, status}
  });
});
```

Getting From a Route to Global Error Handler

- We've set up our global error handler & are throwing errors in routes.
- Once we **throw**, we need to tell Express to send error to global error handler.
- We need to move onto the **next** thing!
- Every route handler can accept **three** parameters, **req, res** and **next!**
- To move to error handler, invoke **next** function inside of our route handlers.

Handling Errors Correctly

```
const ExpressError = require("./expressError")

app.get("/users/:username", function(req, res, next){
  try {
    const user = USERS.find(u => u.username === req.params.username);
    if (!user) throw new ExpressError("Not found!", 404);
    return res.json({ user });
  } catch (err) {
    return next(err);
  }
})

// Global Error Handler goes down here
```

- All logic inside route handlers is wrapped in **try/catch**
- If you want to respond with an error, **throw a new ExpressError**
- In **catch** for route handler, **always** pass error to **next()**

```
try {
  // logic here
} catch (err) {
  return next(err);
}
```

- In **app.js** ensure there is a global error handler at bottom

Handling 404 Errors

If a route does not match, let's make sure to handle that as well.

```
demo/routing-app/app.js

// 404 handler
app.use(function (req, res, next) {
  const notFoundError = new ExpressError("Not Found", 404);
  return next(notFoundError)
});

This goes after all routes, but before the global handler.
```

Putting it all together

```
demo/routing-app/app.js

// 404 handler
app.use(function (req, res, next) {
  const notFoundError = new ExpressError("Not Found", 404);
  return next(notFoundError)
});

// generic error handler
app.use(function(err, req, res, next) {
  // the default status is 500 Internal Server Error
  let status = err.status || 500;
  let message = err.message;

  // set the status and alert the user
  return res.status(status).json({
    error: {message, status}
  });
});

// end generic handler
app.listen(3000, function() {
  console.log('Server is listening on port 3000!');
});
```

Debugging with Express

Current Debugging Process

console.log()

Like **print** in Python, often very useful

Debugging Node

Can also use the Chrome Dev Tools debugger

Start up Node with **--inspect-brk** flag:

```
$ node --inspect-brk sumEvens.js
Debugger listening on ws://127.0.0.1:9229/a98973...
For help, see: https://nodejs.org/en/docs/inspector
```

Open **chrome://inspect** to pull it up in the Chrome Debugger! 🐼🐼🐼

Debugging Express

- With **--inspect-brk** a breakpoint is put on the first line of your app
- Can start with **--inspect** to not stop at first line:

```
$ nodemon --inspect

Debugger listening on ws://127.0.0.1:9229/a98973...
For help, see: https://nodejs.org/en/docs/inspector
```

- Use the **debugger** keyword in code to activate a breakpoint

Open **chrome://inspect** to pull it up in the Chrome Debugger! 🐼🐼🐼

Your Turn!