Express

- Self-described as a "fast, unopinionated, minimalist web framework"
- Meaning, it has enough functionality to make implementing HTTP requests and responses easy while not obscuring the underlying native HTTP functionality

Brief History of Express

- Started in early 2010
- Originally a way to have a small, robust server to make standing up application routing and template rendering easier
- Morphed into a pluggable framework to handle a variety of use cases, but the central philosophy remains

Installation

1. Create and `cd` into your project directory
2. `npm init`
3. `npm install --save express`

Similar Frameworks/Approaches in Other Languages

- Sinatra (Ruby)
- Laravel (PHP)
- ASP.NET Routing (.NET)
- Django (Python)

Routing

- Routing methods are generally of the form `router.METHOD(PATH, HANDLER)`
- `METHOD` defines the kind of request (`GET`, `POST`, etc)
- `PATH` defines what the request handler can respond to
- `HANDLER` is a function holding your logic to handle the request

Defining a Handler

- Handlers are functions applied to the given route that take up to three parameters:
  - **request:** an object housing all of the request information, including what URL was matched, the body of the request (if there was one), the parameters for a request (if there are any), the query string information (if there is any)
  - **response:** an object containing methods for responding to a request
  - **next:** (optional) instead of responding to the request directly,

Routing Example

```
//include express
var express = require('express');
//create an express application
var app = express();

//define a route on `/hello/world`
app.get('/hello/world', function(request, response) {
    console.log('got request for "/hello/world"');
    response.send('hello there!');
});

//have the application listen on a specific port
app.listen(3000, function () {
    console.log('Example app listening on port 3000!');
});
```

Statuses

- Returning HTTP statuses in Express can be done using `response.status(status)`

```
js
app.get('/hello', function(request, response)
{ ... });
//if no routes are matched, return a 404
app.get('*', function(request, response) {
request.status(404).send('uh oh! page not
found!');
});
```

Parameterized Routing

- Parameterized routes contain one or more parts that are variables
- In Express, these variables are denoted on the route with a colon (`:`)
- The variables in the wroute are put onto the `params` member of `request`

Parameterized Routing Example

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

//define a route to greet someone by name, eg /hello/sally
app.get('/hello/:name', function(request, response) {
    console.log(`got request for "/hello/${request.params.name}"`);
    //in the above example, returns "hello sally!"
    response.send(`hello ${request.params.name}!`);
});

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

Routes with String Patterns

- Patterns are useful when you want to catch alternate spellings, typos, or want to apply middleware to many routes
- Routes support:
  - Wildcards (`*`)
  - Character repetition (`+`)
  - Optional Characters (`a?`)
  - Optional Group (`(ab)?`)

Be Careful with Groups

- Because Express relies on a library called `path-to-regex`, groups directly after a leading slash do not work

```
//will throw an error complaining about an invalid regular expression group
app.get('/(hel)?lo', function(req, res) { ... })
```

Multiple Route Names

- Allows you to supply an array of paths for a single handler
- Makes it easy to alias one route for another if they perform the same action

```
app.get(['/hello', '/hi', '/hola'], function(request, response) { ... });
```

Be Careful with Route Ordering

- Consider the following:
```
js
app.get('/hello/:name', function(request,
response) { ... });
app.get('/hello/world', function(request,
response) { ... });
```

- Issuing a call to `/hello/world` will execute the first route it matches, and in this case, `/hello/:name` will fulfill the request!

- How could we fix this problem?

Exercise: Routing

- Create routes to do the following:
  - Accept `apple` or `ale`, returning `"Apple or Ale?"`.
  - Accept the word `whoa` with an aribitrary number of `o`s and `a`s, returning `"I know, right?!"`.
  - Take a first name and last name as parameters, returning a greeting for that user.
  - Take a word as a parameter and returning the word reversed.
  - Add a route that will execute if nothing else is matched,

Query Strings

- Query Strings provide extra information on the end of a url
- Information is in key-value pairs
- Express puts this information into `request.query`

```
http://my-cool-site.com/page?foo=bar&baz=quux
```

- would be translated into this `request.query`:

  ```
  {
  'foo': 'bar',
  'baz': 'quux'
  ```

Exercise: Query Strings

- Add a route to the previous example that returns a friendly greeting for `firstname` and `lastname` query parameters on the route `/hello`.

Middleware

- Middleware adds useful functions after the request is received but before the route is handled
- Logging, authentication, and parsing are all good candidates for being middleware
- Middleware can be appiled to the entire application, routers, or individual routes
  ```js
  var express = require('express');
  var app = express();
  ```

Exercise: Middleware

- Create middleware for the earlier examples to make a log of incoming requests. Include the original route and a timestamp. Have the log write to a file called "log.txt" in your project directory.
- Hint: the original route for the request is on `request.originalUrl`.

Body Parsing

- Express does not handle parsing internally, delegating that responsibility to middleware outside of itself
- `body-parser` is maintained under the Express project: `npm install --save body-parser`
- This handles the request stream and deserialization for you.
- Usage:

```
var express = require('express');
var parser = require('body-parser');
var app = express();

//parses requests with the content type of `application/json`
app.use(parser.json());

app.post('/submit', function(request, response) {
    //if a json payload is posted to `/submit`,
    //body-parser's json parser will parse it and
    //attach it as `request.body`.
    console.log(request.body);
    response.send('request receieved.');
});
```

Exercise: Body Parsing

- Install `body-parser`.
- Use the JSON middleware to parse requests.
- Add a POST handler on `/submit`. Have the method print out the `request.body`.
- If the payload does not have a member called `foo`, return a 404.
- Start your server.
- Use `cURL` to issue a JSON payload to your server on `/submit`. Try it with and without the `foo` member.

Response Encoding - JSON

- So far, all of the data we've seen has been plaintext - what about JSON?
- `res.send()` takes care of setting the response's `Content-Type` to `application/json` automatically if the sent data is an `Array` or `Object`, but if we wanted to be explicit:

```
app.get('/some/route', function(req, res){
    var obj = {'hello': 'there'};
    //get the default JSON replacer from Express (defaults to undefined)
    var replacer = app.get('json replacer');
    //get the default spacing for JSON stringification (defaults to undefined)
    var spaces = app.get('json spaces');
    //stringify your response object
```

Reponse Encoding - HTML

- Again, `res.send()` takes care of setting `Content-Type` to `text/html` automatically if the response is a `String`, but if we wanted to be explicit:

```
app.get('/page', function(req, res){
    res.set('Content-Type', 'text/html');
    res.send('<h1>Hello!</h1>');
});
```

Response Encoding - Plain Text

- This does *not* get taken care of automatically. To send plaintext, set `Content-Type` to `text/plain`:

  ```
  js
  app.get('/page', function(req, res){
  res.set('Content-Type', 'text/plain';
  //will print the string literal to the browser
  instead of rendering HTML
  res.send('<h1>Hello!</h1>');
  });

  ---
  ```