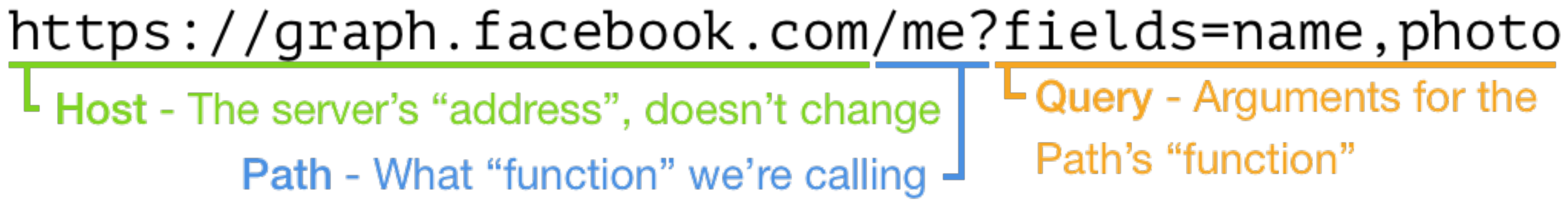
Express #1: Intro and Routing

HTTP Recap

- A few modules ago, we learned about HTTP requests
- They have a URL that consists of a host, path, and query


https://graph.facebook.com/me?fields=name,photo
Host - The server's "address", doesn't change
Path - What "function" we're calling
Query - Arguments for the Path's "function"

- They have a method type of GET (read) POST (create) PUT (edit) or DELETE (remove)
- They can have additional data in headers or the body

Creating Our Own HTTP Server

- There are many ways to create your own HTTP server
- Python has SimpleHTTPServer, Flask, **Django**, and more
- Ruby has Socket, Thin, **Ruby on Rails**, and more
- Javascript has the http module, Koa, **Express**, and more
- In this class, we'll be using Express, as it's the most popular Javascript HTTP server

Intro to Express

- Express was created to wrap the core `http` module and make it easier for people to create an HTTP server with less code
- It's just a module, like any other on NPM, so it's easy to get started
- Just how easy is it? Let's see:

```
// App.js
const express = require("express");
const app = express();

app.listen(3000, function() {
    console.log("Your server is available at localhost:3000!");
});
```

```
# Terminal
npm install --save express
node app.js
```

Intro: Breaking It Down

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

- Here we included the express module. It's just a function that creates a new HTTP server. One node program can create multiple server applications on different ports.

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

- Here we created one of those servers. By default this server does nothing, and is just waiting to be set up. It takes no argument, since all setup will be done on app, not in the `express` function.

```
app.listen(3000, function() {
    console.log("Your server is available at localhost:3000!");
});
```

- Finally, we turned the server on by telling it to `listen` to port 3000. Once the server has started, it fires the callback we passed it. Servers only take in a port, not a URL, because they always listen on `localhost`

Quick Note on Ports

- The `3000` we used earlier was a **port**, which is basically a gateway on the machine to talk through. It could have been any number, but 3000 is a common one used for developers.
- By default, the web communicates on port `80` (HTTP) and port `443` (HTTPS) unless the server specifies differently.
- We won't use those for now since our application will only be communicating locally (on localhost.)
- Try to avoid low number common ports, only one application can listen on one port at a time.

Configuring Your Server

- Right now hitting our HTTP server returns an error, since it hasn't been configured
- The main configuration of a server is adding **routes**, functions that run when certain paths have been hit
- Let's add a route for the homepage, shall we?

```
/* require express, create app */

app.get("/", function(req, res) {
    console.log("Serving up homepage...");
    res.send("Hello!");
});

/* app.listen below */
```

**You have to restart your server to see any changes.*

Route: Breaking It Down

```
app.get(/* ... */)
```

- The app object has four main functions for each of the method types. This listens only for `GET` requests.

```
app.get("/", /* ... */)
```

- The first argument is what path the client will need to request to see this page. `"/"` is the homepage.

```
app.get("/", function(req, res) {
    console.log("Serving up homepage...");
    res.send("Hello!");
});
```

- The function you provide is a callback for when a client requests this page. The `req` object contains information about the client's **req**uest to the server. The `res` object provides ways of customizing the **res**ponse we send from the server.

Try It For Yourself

- Create 2 more routes that use different paths
- Make them output different things from each other using `res.send`
- Open them up in your browser and make sure they work

Rendering HTML

- Using `res.send` can come in handy, but usually we want to send a lot of HTML stored in another file
- Rather than implementing our own file opening and sending code, Express provides us with `res.render` to display views
- This takes in a `path` argument that looks for an file in a `views` folder, relative to the project's directory

```
app.set("view engine", "ejs");

app.get("/", function(req, res) {
    res.render("homepage");
});

app.get("/gallery", function(req, res) {
    res.render("gallery");
});
```

```
./my-express-project
├── package.json    # Standard package.json
├── app.js          # Where the server is configured
└── views           # Where we store the views
    ├── homepage.ejs    # HTML that is rendered at / (or no path)
    └── gallery.ejs     # HTML that is rendered at /gallery
```

What are the view engine and ejs files?

- Express uses one of many "view engine"s to render HTML, EJS is the one we set in the code above
- Because HTML on its own can't render dynamic content, Express only works if you use one of these engines
- We'll be covering view engines in a future lecture, but just know for now that an EJS file can contain HTML just like normal
- However, we'll need to install the `ejs` module so that Express can use it

```
npm install --save ejs
```

Static Files

- While sometimes we want to do clever routing to load files, some things should just match our folder structure
- Typically assets are this way, things like images, javascript, css etc.
- Instead of exposing our whole project folder over HTTP, leaking our code in to the world, we specify a folder for static assets
- We then configure that on our app using `express.static()`
- This should be done *before custom routes*, so that it doesn't get overridden

```
// require express, create app

app.use(express.static("assets"));

// configure routes, app.listen
```

Static Files: Breaking It Down

```
app.use(/* ... */)
```

- This tells our application to `use()` some common functionality. This is great for applying complicated pre-configured settings to our app, without writing all the code ourselves.

```
app.use(express.static(/* ... */));
```

- Calling `express.static()` returns one of those configurations, which designates a route to serve up static files, as they are, no modifications or logic.

```
app.use(express.static("assets"));
```

- And finally, we pass in an argument for what folder we want to keep our static files in. Files in this folder will be available through the server, so now, `localhost:3000/js/script.js` will serve up `/path/to/my-project/assets/js/script.js`.

Exercise: Convert Portfolio page

- Remember your portfolio site from the beginning of class? Let's expressify it!
- Move your HTML files to a `views` folder.
- Create routes for each of them that `res.render` the HTML files
- Move your static assets in to the folder you specified with `express.static()`
- Fix the `src` and `url` paths to reference your static assets folder

Additional Resources

- Express's Getting Started guide - Straight from the horse's mouth, a similar walkthrough on how to start an Express project
- Express's Complete API - If you want to dive deep on `express`, app, `req`, or `res`, 100% of their functionality is laid out here.