Routing Functions and Routers

Outline and Learning Objectives

- **Routing Functions Overview:**
  - to understand the design and behaviour of Routing Functions, compared to Middleware
- **Typologies of Routing Functions:**
  - to master the different typologies of routing functions, parameters and query strings
- **Subapplications (Routers):**
  - to understand how to structure larger apps with Express.js and Subapplications
- **Serving Static Files**
  - to understand more advanced aspects and scenarios when serving static files

- **Suggestions for Reading**

Routing Functions Overview

Routing

- Like **Middleware**, **Routing** breaks the **one big request handler function into smaller pieces**.

- **(When reached)** Routing Functions are **executed on the base of conditions (Client Requests):**
  - **URL requested** (e.g., /homepage)
  - **HTTP Method** (e.g., GET, POST, PUT, DELETE, …)
- Routing functions are **executed in the order they are added to the stack (but conditionally, as above)**
- When one **Routing** Function **finishes**, usually **the response is returned**

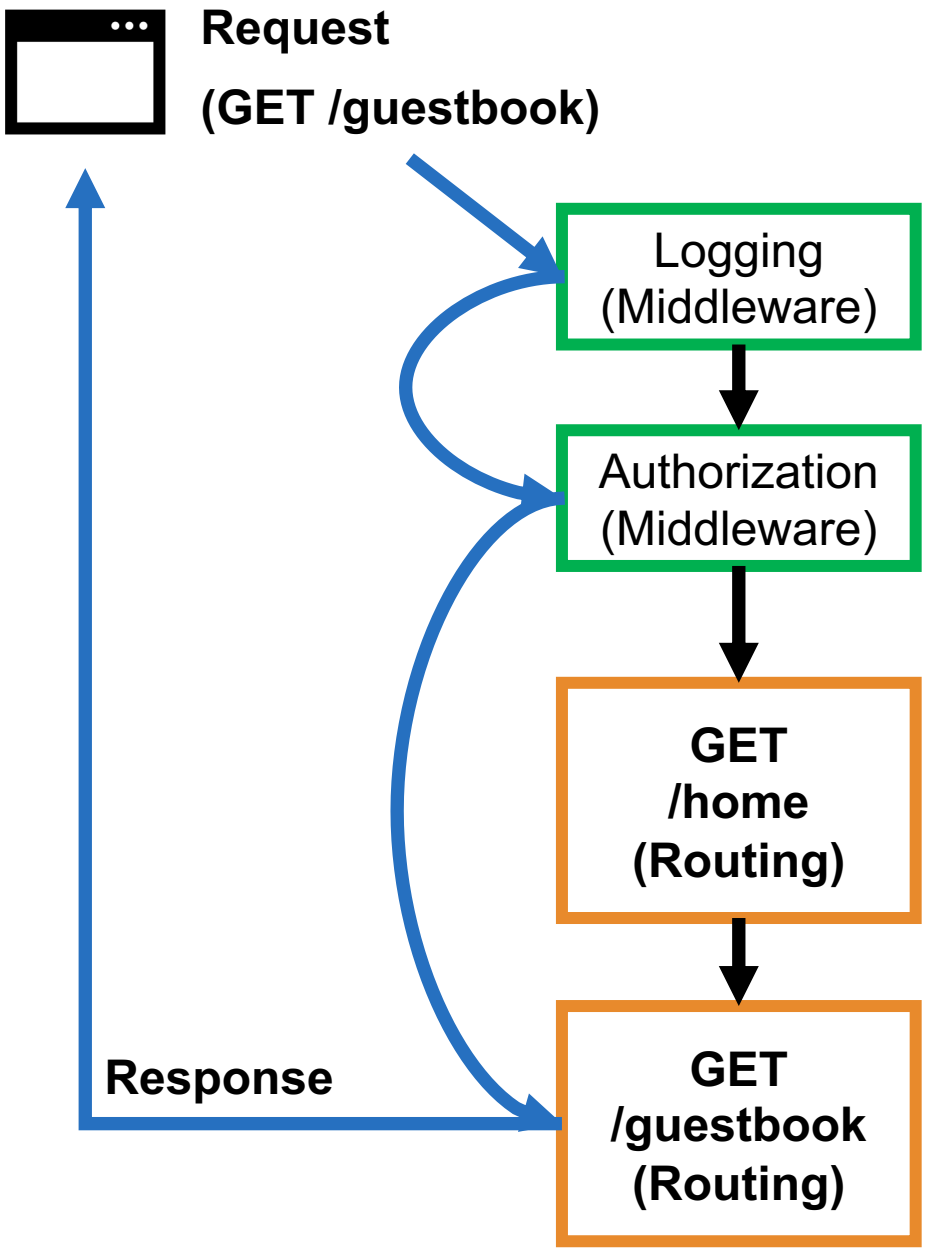
- **For example**, you might build a **web page with a homepage and a guestbook**:
  - when the browser sends an **"HTTP GET"** to the **homepage URL**, Express should **send the homepage**.
  - when the browsers asks for the **guestbook URL**, it should send them the **HTML for the guestbook**.

Routing Vs. Middleware and Example

- **[Common Objective]** rather than one monolithic request handler function, the aim is to have **several request handler functions** that each deal with **a small chunk of the work**.

- **[Middleware] (When reached)** Middleware Functions are **always executed**
- Middleware functions are **executed in the order they are added to the stack**
- When one **Middleware finishes**, usually Express will continue onto **next**

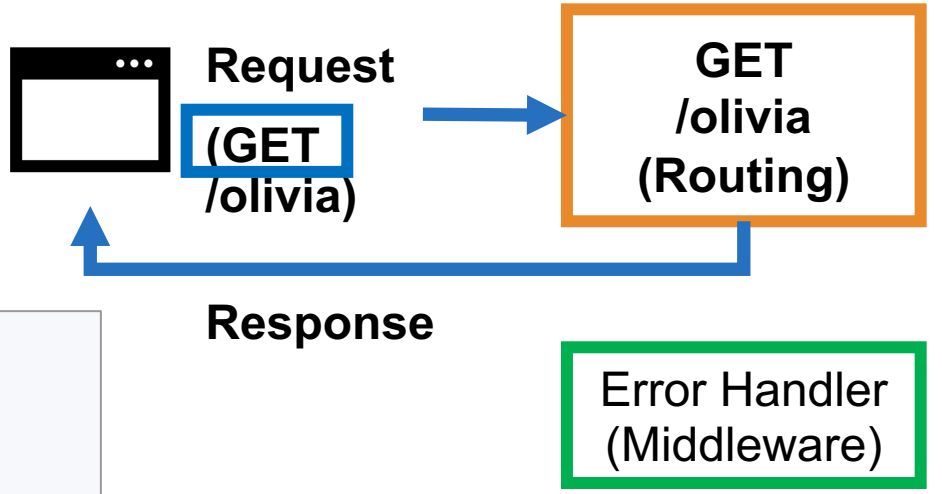
- **[Routing] (When reached)** Routing Functions are **executed on the base of conditions (Client Requests):**
  - **URL requested** (e.g., /homepage)
  - **HTTP Method** (e.g., GET, POST, PUT, DELETE, …)
- Routing functions are **executed in the order they are added to the stack (but conditionally, as above)**
- When one **Routing** Function **finishes**, usually **the response is returned**

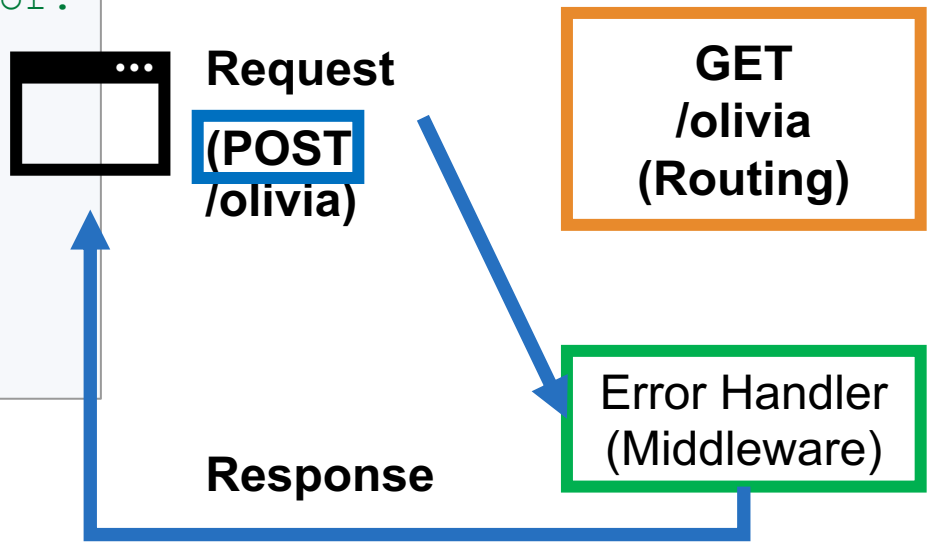


First Routing Function: Example

- When you visit a web address like `example.com/olivia`, you are asking for the **verb** `GET` and the **URL** `/olivia`

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

// Routes GET requests to /olivia to the request handler
app.get("/olivia", function(request, response) {
   response.send("Welcome to Olivia's homepage!");
});

// If you load something other than /olivia, serves a 404 error.
// If it is not a GET request, also servers a 404 error.
app.use(function(request, response) {
   response.status(404).send("Page not found!");
});

// Starts the server on port 3000
app.listen(3000);
```

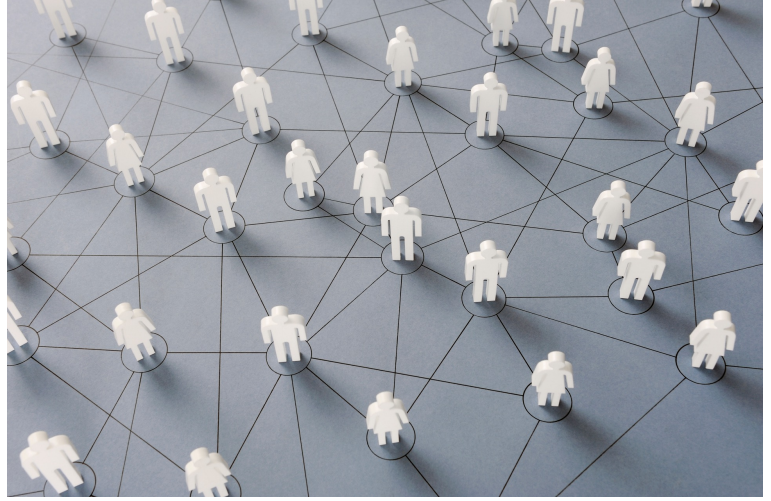




Typologies of Routing Functions (Part 1)

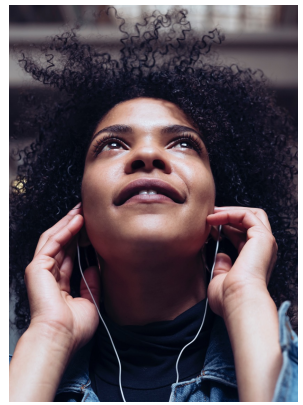
1. Routing Functions with Parameters
2. Routing Functions with Query String Arguments

Get User by Id Parameter: Example

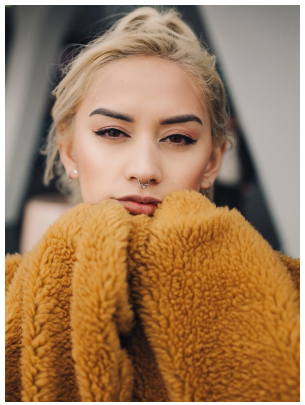


- Imagine you need to make a website whose user has a numeric ID such as 1, 2, 3, etc.
- You want the URL for **user #1** to be `/users/1`, **user #2** `/users/2`, and so on.
- Rather than **define a new route for every user** (which would be **bad practice**),
  - you can **define one route that starts with** `/users/` and then has an **ID parameter**.

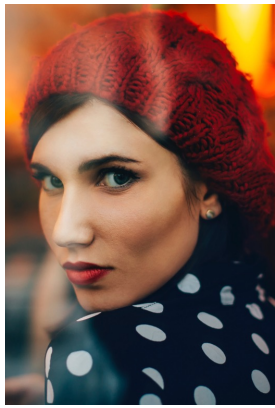
```
// Matches requests coming into /users/123 and /users/olivia
app.get("/users/:userid", function(req, res) {
   // Convert userid into an integer
   var userId = parseInt(req.params.userid, 10); // base 10
   ...
});
```



/users/1



/users/2



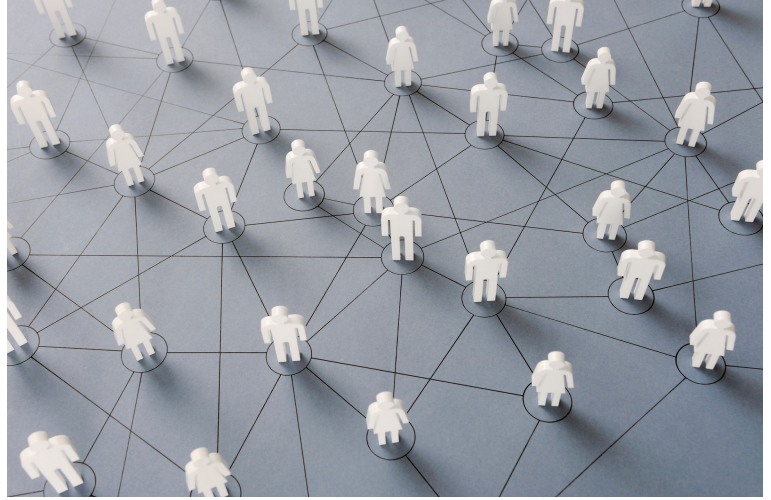
/users/3

- Colon "**:**" followed by the parameter name:
  - `:PARAMETER-NAME`
- Then, you can **retrieve the parameter from the request**
- The **route will not match** `/users/` or `/users/123/posts`
- **It will also match** `/users/cake` and `/users/horse_ebooks`

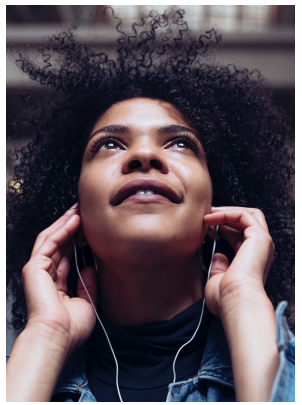
How to manage this -> Next Slide

Parameters and Regular Expressions

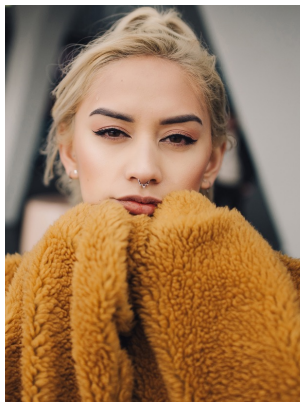
- **2 solutions**:
  - **[Middleware Logic]** Implements checks on the param received in the logic of the Middleware or, better
  - **[RegEx]** Implement the Middleware to match with a Regular Expression (RegEx)



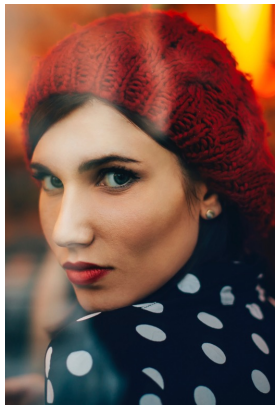
```
// Matches requests coming into /users/123 , NOT /users/olivia
app.get(/^\/users\/(\d+)$/, function(req, res) {
    // Convert userid into an integer
    var userId = parseInt(req.params[0], 10); // base 10
    res.send("userId: " + userId);
});
```



/users/1



/users/2



/users/3

- You use a RegEx:
  - `/^\/users\/(\d+)$/`
- Then, you can **retrieve the parameter from the params array**
- The **route will not match** `/users/` or `/users/123/posts`
- The **route will not match** `/users/cake` and `/users/horse_ebooks`

Typologies of Routing Functions (Part 2)

1. Routing Functions with Parameters
2. Routing Functions with Query Arguments

Managing Query Arguments

Another common way to dynamically pass information in URLs is to use **query strings**

If you searched for "**javascript-themed burrito**" on **Google**

- the **URL** would be like: `https://www.google.com/search?q=javascript-themed%20burrito`

This is passing a query. If Google were written in Express, it might handle a query like this:

```
app.get("/search", function(req, res) {
    if (req.query.q === "javascript-themed burrito")
{
        res.send("Burrito search performed");
    } else {
        res.send("Another query and/or parameter");
    }
});
```

- the `q` in the `req.query.q` needs to match the `...?q=...` in the query string `https://www.google.com/search?q=javascript-themed%20burrito`

Query Arguments with Multiple Parameters

- Multiple parameters in a query string are specified as:
  - http://localhost:3000/multipleparamssearch?param1name=value&param2name=value
- For instance:
  - http://localhost:3000/multipleparamssearch?q=javascript-themed%20burrito&distance=5km
- & is the **parameters separator**; the following is an **example with 2 parameters**

- Example: **Burrito in the range of 5 KM** ☺
- Route with **2 parameters**: q and distance

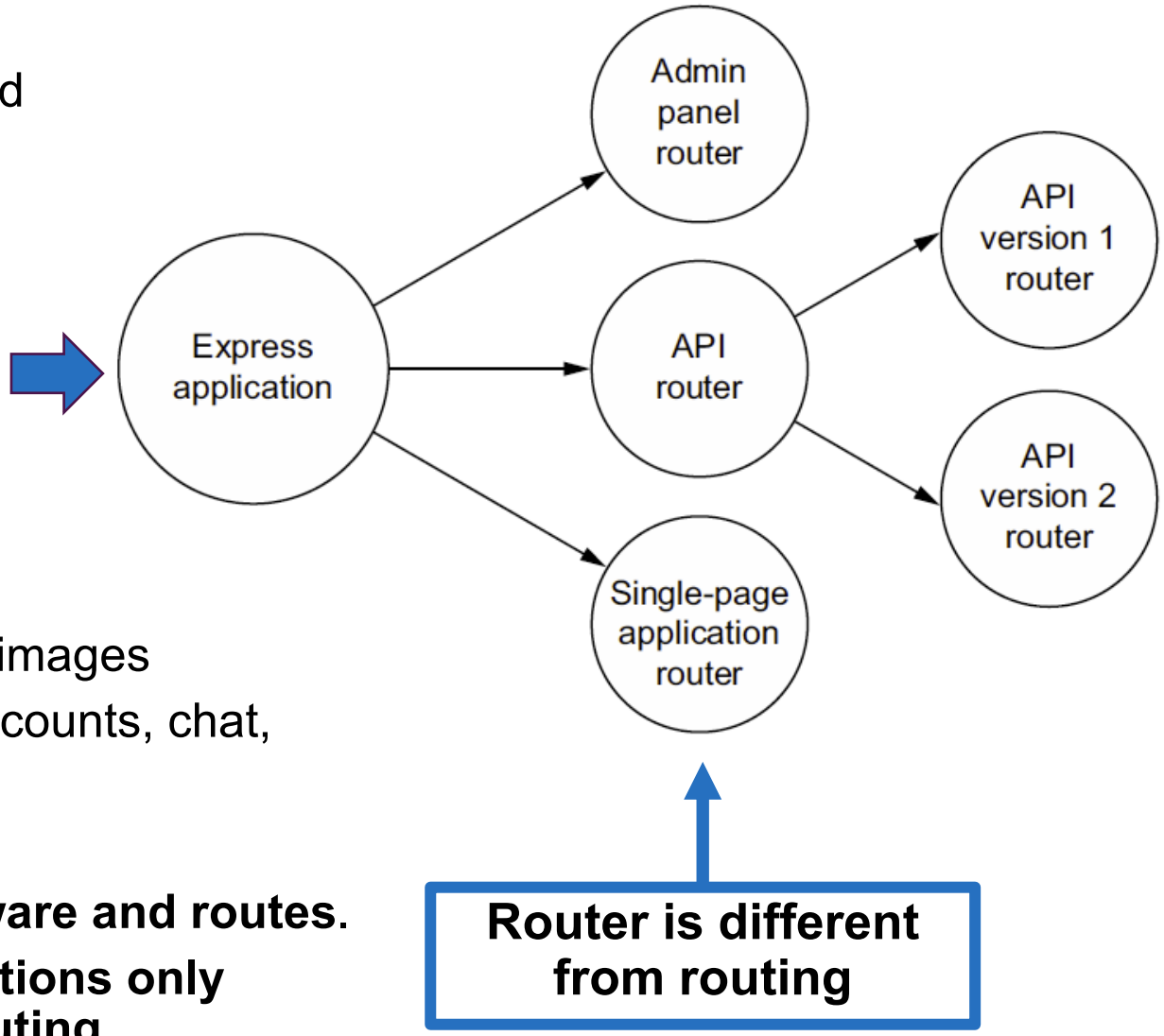
```
app.get("/multipleparamssearch", function(req, res) {
    if (req.query.q === "javascript-themed burrito") {
        if (req.query.distance === "5km") {
            res.send("Restaurant A");
        } else {
            res.send("Distance Param not found or no results");
        }
    } else {
        res.send("Another query and/or parameter");
    }
});
```

Subapplications (Routers)

Express.js and Subapplications (Routers)

Subapplications (Routers)

- Express allows you to define routers that can be used to **break up larger applications**.
  - It allows you to further **compartmentalize your app into smaller pieces**.
- You might have an administration panel in your app, and that can function differently from the rest of your app.



Using routers to split up your app

- **As your app grows, so will the number of routes**
  - It may start with routes for static files and for images
  - Later you need to add new routes for user accounts, chat, forums, and so on
- **Express 4 introduces routers for this issue**
  - A **router** is an **isolated instance of middleware and routes**.
  - Routers can be thought of as **"mini" applications only capable of performing middleware and routing**.
- **Every express application has a built-in app router**.

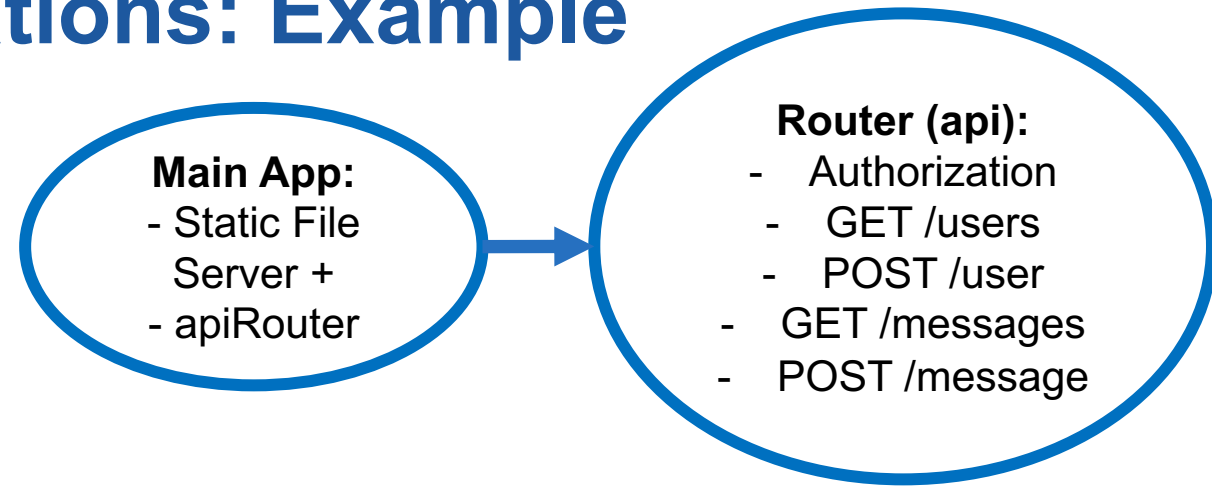
Using Subapplications: Example

- Main App File

```
var express = require("express");
var path = require("path");

// require your router
var apiRouter = require("./routes/api_router");

var app = express();
var staticPath = path.resolve(__dirname, "static");
app.use(express.static(staticPath));

// use your router
// any URL that starts with '/api' will be
// sent to 'apiRouter',
// such as '/api/users' and '/api/message'
app.use("/api", apiRouter);

app.listen(3000, "0.0.0.0", function() {
   console.log("App started on port 3000");
});
```

- This is to force Node.js to use **IPv4** (Node.js by default uses **IPv6**)

- **Any URL that that starts with** `/api` **will be sent to** `apiRouter`, for example:
  - `/api/users`
  - `/api/message`
  - …



- The Router File (`routes/api_router.js`)

```
var express = require("express");
var ALLOWED_IPS = ["127.0.0.1", "123.456.7.89"];
var api = express.Router();
api.use(function(req, res, next) {
   var userIsAllowed = ALLOWED_IPS.indexOf(req.ip) !== -1;
   if (!userIsAllowed) {
      res.status(401).send("Not authorized!");
   } else {
      next();
   }
});
api.get("/users", function(req, res) { /* ... */ });
api.post("/user", function(req, res) { /* ... */ });
api.get("/messages", function(req, res) { /* ... */ });
api.post("/message", function(req, res) { /* ... */ });
module.exports = api;
```

Serving Static Files

Serving Static Files: More Advanced Features

Previously:

- **[Using Midldeware]** serving static file with a `public` **Folder**

```
// Sets up the path where your static files
//are
var publicPath = path.resolve(__dirname,
                              "public");
// Sends static files from the publicPath
//directory
app.use(express.static(publicPath));
```

- Calling this with "**http://localhost:3000/cool.txt**", will return the file "**cool.txt**" (if it is in the **public** folder of the server)
- Calling this with "**http://localhost:3000/image/cat.png**", will return the file "**cat.png**" (if it is in the **images** folder of the server)
- Calling this with "**http://localhost:3000/public/cool.txt**", will return the file "**cool.txt**" (if it is in the **public** folder of the server)

- **IMPORTANT:** as you can see from the example of the images, **local folder (images)** and the **path indicated in the middleware (/image) does not need to coincide**

[Using Rooting] Changing the File Path

- You can choose to serve the files in a different location based on the request url
- For example, all the pictures are in a **folder** `images`
  - Not the `public` **folder**

```
var imagePath = path.resolve(__dirname, "images");
app.use("/image", express.static(imagePath));
```

[Using Rooting] Multiple File Paths

- You can set up **multiple URLs** for **different static file folders**
- For example, we have both the `public` and `images` folders for static files
  - And the files in there can be requested with different URLs

```
var publicPath = path.resolve(__dirname, "public");
var imagePath = path.resolve(__dirname, "images");

app.use('/public', express.static(publicPath));
app.use('/image', express.static(imagePath));
```

Suggestions for Reading

Reading

Chapter 5 of the **“Express in Action”** textbook

Questions?