

Express | Introduction


Learning Goals

After this lesson you will be able to:

- Understand the idea of a framework
- Understand the two most common HTTP verbs: `GET` and `POST`
- Use the Express framework to create a basic web application
- Incorporate static assets (CSS and images) into your Express application
- Keep your HTML code for your Express app in separate files

Introduction

In this lesson, we will learn how a **framework** like Express can help us make our back end code more organized. Before we can get into the real capabilities of the back end, we must first learn how to set up a basic Website using back end technologies. That's exactly what we will be learning here: **how to make a basic Website** using Node.js and Express.

 This kind of Website **doesn't need Express or back end**. It would be better to make a Website like this with plain old HTML, CSS and JavaScript.

We are only making these simple Websites to lay the foundation of how to use Express so we can do more appropriate back end tasks later.

Web Frameworks

Why would I use a *framework*?

[Web Frameworks](#) or web application framework (WAF) is a software framework that is designed to support the development of web applications including web services, web resources and web APIs. Web frameworks aim to alleviate the overhead associated with common activities performed in web development.

For example, many web frameworks provide:

- **Libraries for database access** (for saving your data permanently)
- **Templating frameworks** (for making dynamic HTML that changes between users)
- **Session management** (for keeping track of users)
- **Often promote code reuse** (for keeping your code DRY)

Though they often target development of dynamic websites they are also applicable to static websites.

We can build a full-blown web application using just Node.js, Ruby, Python or any other backend language. Over time though, people have developed *frameworks* on top of these different languages for a few reasons:

- We do many of the aforementioned tasks for every web application, and it's tedious to set up.
- We need a way to structure our code, all of our complex logic can build up over time, leaving us with a mess.
- We benefit from other people's hard work. If you're using your own framework, then every bug is yours to fix. With a community surrounding the framework, we get some help in this regard.

Express Web Framework



[ExpressJS](#) is the most commonly used framework in the Node.js ecosystem. Every Node.js player has heard of it and is using it with or without noticing.

It's currently on its 4th generation, and there are quite a few [Node.js frameworks](#) built based upon it or inspired by its concepts.

Express.js, or simply Express, is a web application framework for Node.js, released as free and open-source software under the MIT License.

It is designed for building web applications and APIs. It is the de facto standard server framework for Node.js. The original author described it as a relatively minimal with many features available as plugins.

Express is the backend part of the MEAN stack, together with the MongoDB database and the Angular frontend framework.

More HTTP - HTTP Verbs

Before we get into our first Express code, we need to add a little to our knowledge of HTTP. We need to talk about something called **HTTP verbs**.

When a client sends an HTTP request to a server, aside from specifying the URL the client also has to specify an extra piece of text called an **HTTP verb** (also known as an *HTTP method*). This **HTTP verb** has to be one of a list of valid words in the HTTP specification. The verbs are designed for clients to **communicate the intention of the request** to servers. Servers software is programmed to consider the verb when deciding what to do with a request.

The most common two verbs are `GET` and `POST`. Most of the requests your browser makes on a daily basis are `GET` requests, to retrieve some kind of information. For example, we make a `GET` request to `https://www.netflix.com/browse`, and the server gives us back an HTML page. The server decides whether a request using that verb and that URL is possible.

We need to know about these verbs because when we program our own back end, we decide which **HTTP verbs** work with which URLs.

GET

`GET` requests:

- Should be used only to retrieve data from a server
- Can be cached
- Remain in the browser history
- Can be bookmarked
- Should never be used to send sensitive data (more on this later)

- Have length restrictions (you can't send giant files)

Example

Type google.com in your URL bar and hit enter. You just made a `GET` request!

POST

A `POST` request is for *sending data to the server*.

`POST` requests:

- Should be used to send data to the server
- Are never cached
- Do not remain in the browser history
- Cannot be bookmarked
- Are better for sensitive data (again, more on this later)
- Have no restrictions on data length (you can send big files)

Example

When you enter information into a login form on facebook, and hit submit you are making a `POST`. You're saying to Facebook's servers: "Here's my login info, log me in please".

Also, when making banking transactions they are often handled through `POST` requests. This is to prevent you from withdrawing (or crediting) your money over and over by accidentally repeating transactions. You wouldn't want to pay your bills twice, would you?

Express Hello World

[ExpressJS](#) is a web framework built on Node.js that has functions that closely resemble the request-response process.

Let's get set up with our first Express app.

1 | Create a folder called `express-hello-world`.

```
$ mkdir express-hello-world
$ cd express-hello-world
```

2 | Install Express with NPM and save it as a dependency in our project, then create a file to run our server named `app.js`

```
$ npm init
(Hit return until prompted to type "yes")

$ npm install express --save
$ touch app.js
```

3 | Write our `app.js` server

We have to first require Express so we can use it in our app.

The `express()` function creates an Express application.

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

// We create our own server named app
// Express server handling requests and responses
const app = express();
```

4 | Set up a Route

Web Apps have many different pages, and we need to tell our server what to do when it receives a request. Normally a server identify requests with two parameters:

- **URLs**

In back end, we only consider the part of the URL that comes after the the domain. Examples:

- If the full URL is `example.com/profile` we refer to it as `/profile`
- If the full URL is `example.com/blogs` we refer to it as `/blogs`
- If the full URL is `example.com` we refer to it as `/`

- **HTTP Method:** `get`

For now, we will stick to `GET` requests. We will talk more about `POST` in a later lesson.

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

// our first Route
app.get('/', (request, response, next) => {
  console.log(request);
  response.send('<h1>Welcome Techgrounder. :)</h1>');
});
```

Notice, each route will accept a *callback*. This is the function that will be called when someone makes a request to `/`.

- **app:** Our express server
- **get:** the **HTTP Verb** needed to access this page
- **/:** the route that the User will type into the URL bar
- **request:** An *object* containing information about the request, such as the headers. More on this later.
- **response:** An *object* containing information about the response, such as headers and any data we need to send to the client.
- **next:** We will use this later to handle errors. Leave it there for now.

The three parameters of the `request`, `response` and `next` will always be passed to the callback function for any route. The `response` object has methods that allow us to control what we send the client. In this case, we are using the `send()` method that just sends a string.

5 | Start the Server!

Tell our server to *continuously listen for requests* on port 3000. You can optionally provide a callback to do something once the listening is set up.

```
// Server Started
app.listen(3000, () => {
  console.log('My first app listening on port 3000!')
});
```

Our first Express code

```
// Require Express
const express = require('express');

// Express server handling requests and responses
const app = express();

// our first Route:
app.get('/', (request, response, next) => {
  response.send('<h1>Welcome Techgrounder. :)</h1>');
});

// Server Started
app.listen(3000, () => {
  console.log('My first app listening on port 3000!');
});
```

To view your first web app, we must run the server with node. Remember that because we executed `app.listen()`, the program will run until we stop it manually.

```
$ node app.js
```

And visit localhost:3000!



To stop the server, type `CONTROL+C` in the terminal

nodemon

We are about to make a lot of changes to our `app.js`. Remember that every time you change something in your server program, you have to stop it with `CONTROL + C` and run it again with `node app.js`. Let's fix that annoyance right now!

There is another awesome npm package called **nodemon** that can help us reload our application automatically every time we save a change in our JavaScript. First, let's start by installing it:

```
$ npm install nodemon --global
```

We are installing **nodemon** with the `--global` option, which means we are installing it as a command in the terminal. This will allow us to run our application using the `nodemon` command instead of `node`.

```
$ nodemon app.js
```

Once you have your `app.js` running with **nodemon**, you will see it reload automatically in the terminal **every time you save**. No need to `CONTROL + C` anymore!

Static Files

Static files are things like images, CSS, and client-side JavaScript that are sent directly from the server to the browser as is.

Express has built-in support for serving these kinds of files.

Typically, these files are saved in a folder called `public`:

```
$ mkdir public
```

Let's create a folder for images, and download a cool cat picture to that folder:

```
$ mkdir public/images
$ curl -o public/images/cool-cat.jpg http://wallpapercave.com/wp/X7VjxFk.jpg
```

Inside of Express, we have to tell our server to serve static files from the `public` directory:

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

// Make everything inside of public/ available
app.use(express.static('public'));

// our first Route:
app.get('/', (request, response, next) => {
  // ...
}
```

Visit localhost:3000/images/cool-cat.jpg and you will see your image! Notice that the URL **doesn't include the** `public` part. This is important when you are linking your HTML to these external files.

One of the benefits of using static assets is that we can incorporate them into our HTML for the browser to read.

Create a stylesheet:

```
$ mkdir public/stylesheet  
$ touch public/stylesheet/style.css
```

Add the following styles to `style.css`:

```
body {  
  color: blue;  
  background-color: bisque;  
}
```

Create a new route:

```
// ...  
  
app.get('/cat', (request, response, next) => {  
  response.send(`  
    <!doctype html>  
    <html>  
      <head>  
        <meta charset="utf-8">  
        <title>Cat</title>  
        <link rel="stylesheet" href="/stylesheet/style.css" />  
      </head>  
      <body>  
        <h1>Cat</h1>  
        <p>This is my second route</p>  
          
      </body>  
    </html>  
  `);  
});  
  
// ...
```

Now visit localhost:3000/cat!

Again, notice the URLs we use when linking to these files:

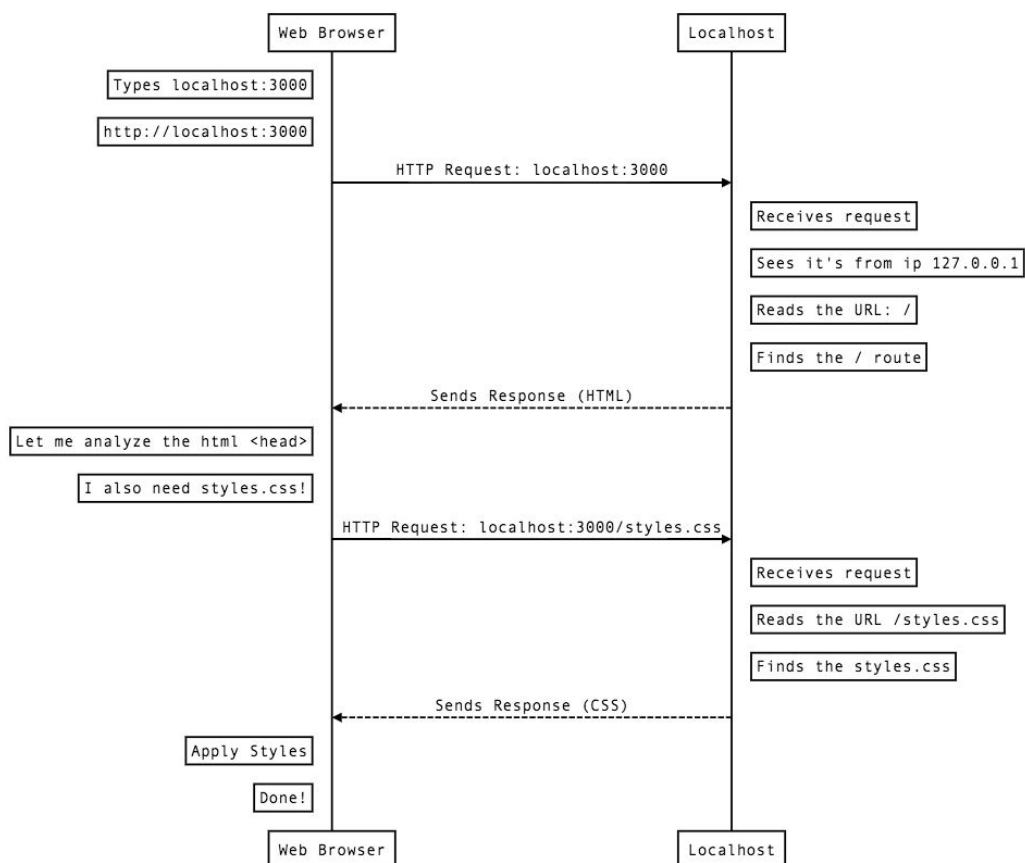
1. `public/images/cool-cat.jpg` is linked as `/images/cool-cat.jpg`
2. `public/stylesheets/style.css` is linked as `/stylesheets/style.css`

Linking Static Files

Some **important observations** when you are linking static files in Express:

- Leave out the `public` part
- Always start the link with `/`
- Never hardcode `localhost:3000`

Statics Assets Request-Response Flow



Separate HTML

In the previous example, we started adding a more realistic HTML structure to our `send()` method. As you can see, it can get very messy to put a bunch of HTML inside a JavaScript string. It has a few downsides:

1. **Bad readability** - It makes our routes very long, making them harder to read.
2. **No syntax options** - We don't benefit from syntax highlighting and the snippets that we get in an HTML file.
3. **Bad separation of concerns** - Ultimately, we want to separate the logic of our back end (the routes and their functionality) from the appearance of our application (the HTML and CSS that users see). In this case, mixing the two leads to bad code.

To solve these problems, we have the ability to create separate HTML files for each of our pages' content. This may sound similar to what we used to do in Module 1 but the key difference is that **the URL of the page doesn't have to match the HTML file name**. Let's see how this is done.

We start by creating a folder for our HTML files and then the files themselves.

```
$ mkdir views
$ touch views/home-page.html
$ touch views/cat-page.html
```



In this case, `views/` is referring to the **presentation code**, the code that the users see (thus the term *views*). We will discuss the name of the `views/` folder in a future lesson.

- The `home-page.html` file will contain the content for our `/` route.
- The `cat-page.html` will contain the content for our `/cat` route.

Starting with `home-page.html`:

```

<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Welcome Techgrounder</title>
    <link rel="stylesheet" href="/stylesheets/style.css" />
  </head>
  <body>
    <h1>Welcome Techgrounder. :)</h1>

    <a href="/cat">See Cat Page</a>
  </body>
</html>

```

Now cat-page.html:

```

<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Cat</title>
    <link rel="stylesheet" href="/stylesheets/style.css" />
  </head>
  <body>
    <h1>Cat</h1>
    <p>This is my second route</p>
    

    <a href="/">Back to Home</a>
  </body>
</html>

```

Notice again how we are linking between the pages using the route URLs:

- / for the home page

- `/cat` for the cat page.

How are the HTML files connected to the routes? That's our next change.

In `app.js` we modify both our routes to refer to the HTML files:

```
// ...
// our first Route:
app.get('/', (request, response, next) => {
  response.sendFile(__dirname + '/views/home-page.html');
});

// cat route:
app.get('/cat', (request, response, next) => {
  response.sendFile(__dirname + '/views/cat-page.html');
});
// ...
```

- Here we are using a new method of `response.sendFile()`. The `sendFile()` method allows us to respond with the contents of a file. It's an alternative to `send()` which only allows us to send a string directly.
- `__dirname` (two underscores) refers to the folder in which our `app.js` is located. Try to `console.log(__dirname)` so you can see it's value. If we don't specify the **complete path to the HTML file** our `sendFile()` will fail.



Don't get confused between the name of the HTML file and the URL. **The URL is determined by the route:** `app.get('/cat', ...)` means that we need to go to `localhost:3000/cat`. Through the `response` we connect to the route's URL to the HTML file: `response.sendFile(__dirname + 'views/cat-page.html');`.



Making these connections is what back end is all about.



Couldn't we have placed the HTML files in `public/` and referred to them directly like `localhost:3000/cat-page.html`? Yes! But we will see later that this structure of having the **route refer to its HTML** will be useful later.

Final version of our first Express app's `app.js`:

```
// Require Express
const express = require('express');

// Express server handling requests and responses
const app = express();

// Make everything inside of public/ available
app.use(express.static('public'));

// our first Route:
app.get('/', (request, response, next) => {
  response.sendFile(__dirname + '/views/home-page.html');
});
```

```
// cat route:
app.get('/cat', (request, response, next) => {
  response.sendFile(__dirname + '/views/cat-page.html');
});

// Server Started
app.listen(3000, () => {
  console.log('My first app listening on port 3000!');
});
```

Summary

In this lesson that a **framework** is code that we can use to organize our back end applications better (instead of starting from scratch every time). We also learned how to build a *super* basic web app with Express. In that app, we incorporated static files like CSS and images and we separated our HTML from our application's main JavaScript file to keep things more organized.

HTTP and web concepts are almost a language on their own. We're going to continue to reinforce and use the different HTTP concepts, including HTTP verbs.