



Node Basics

Node.js is one of the biggest explosions in the past few years. Having the ability to run JavaScript (the client-side language many are familiar) on the server is an enticing notion.

Front-end developers that know JavaScript well can easily expand their knowledge to know back-end server-side code.

Node is built on Google Chrome's V8 JavaScript runtime and sits as the server-side platform in your [MEAN](#) application. So what does that mean? In a LAMP stack you have your web server (Apache, Nginx, etc.) running with the server-side scripting language (PHP, Perl, Python) to create a dynamic website. The server-side code is used to create the application environment by extracting data from the database (MYSQL) and is then interpreted by the web server to produce the web page.

When a new connection is requested, Apache creates a new thread or process to handle that request, which makes it **multithreaded**. Often you will have a number of idle child processes standing by waiting to be assigned to a new request. If you configure your server to only have 50 idle processes and 100 requests come in, some users may experience a connection timeout until some of those processes are freed up. Of course there are several ways to handle this scalability more efficiently, but in general Apache will use one thread per request, so to support more and

more users you will eventually need more and more servers.

This is where Node.js shines. Node is an **event driven language** that can play the same role as Apache. It will interpret the client-side code to produce the web page. They are similar in that each new connection fires a new event, but the main distinction comes from the fact that **Node is asynchronous and single threaded**. Instead of using multiple threads that sit around waiting for a function or event to finish executing, Node uses only one thread to handle all requests. Although this may seem inefficient at first glance, it actually works out well given the asynchronous nature of Node.

Why Node?

Node let's us build real-time open APIs that we can consume and use with our applications. Transferring data or applications like chat systems, status updates, or almost any other scenario that requires quick display of real-time data is where Node does its best.

Some Example Node Uses

- Chat client
- Real-time user updates (like a Twitter feed)
- RSS Feed
- Online Shop
- Polling App

Now that we have a brief overview of Node, let's dive in and build two applications:

- A very simple app to show the basics of starting a Node project
- A more fully-fledged application where we display our latest Instagram pictures

Getting Started

Installing Node

Make sure you have [Node](#) and npm (Node's package manager) installed. npm comes bundled with Node so as long as you install Node, you'll have access to npm.

Once you've installed Node, make sure you have access to Node and npm from the command line. Let's verify installation by checking version numbers. Go into your command line application and type `node -v` and `npm -v`. You should see the following:

```
$ node -v
v6.0.0

$ npm -v
3.8.6
```

If you have trouble with the installation process, try restarting your computer and making sure that

Node is in your PATH. Check the [Node installation page](#) and [npm install page](#) for more troubleshooting.

Now that we have Node and npm installed, let's get to our first app!

A Very Simple Node App

Code for this example found here: <https://github.com/scotch-io/node-booklet-code/tree/master/app1>

Node applications are configured within a file called `package.json`. You will need a `package.json` file for each project you create. This file is where you configure the name of your project, versions, repository, author, and the all important dependencies. Here is a sample `package.json` file:

```
{
  "name": "node-app",
  "version": "1.0.0",
  "description": "The code repository for the Node booklet.",
  "main": "server.js",
  "repository": {
    "type": "git",
    "url": "https://github.com/scotch-io/node-booklet-code"
  },
  "dependencies": {
    "express": "latest",
    "mongoose": "latest"
  },
  "author": "Chris Sevilleja",
  "license": "MIT",
  "homepage": "https://scotch.io"
}
```

That seems overwhelming at first, but if you take it line by line, you can see that a lot of the attributes created here make it easier for other developers to jump into the project. We'll look through all these different parts later in the book, but here's a very simple `package.json` with only the required parts.

```
{
  "name": "node-booklet-code",
  "main": "server.js"
}
```

These are the most basic required attributes.

main tells Node which file to use when we want to start our applications. We'll name that file `server.js` for all of our applications and that will be where we start our applications.

For more of the attributes that can be specified in our `package.json` files, here are the [package.json docs](#).

Initialize Node App

The `package.json` file is how we will start every application. It can be hard to remember exactly what goes into a `package.json` file, so `npm` has created an easy to remember command that lets you build out your `package.json` file quickly and easily. That command is **`npm init`**.

Let's create a sample project and test out the `npm init` command.

1. Create a folder: `mkdir awesome-test`
2. Jump into that folder: `cd awesome-test`
3. Start our Node project: `npm init`

It will give you a few options that you can leave as default, blank, or customize as you wish. For now, you can leave everything default except for the main (entry point) file. Ours will be called `server.js`.

You can see that our new `package.json` file is built and we have our first Node project!

Since we have a `package.json` file now, we can go into our command line and type `node server.js` to start up this Node app! It will just throw an error since we haven't created the **`server.js`** file that we want to use to begin our Node application. Not very encouraging to see an error on our first time starting a Node server! Let's change that and make an application that does something.

Now we will need to create the `server.js` file. The only thing we will do here is `console.log` out some information. **`console.log()`** is the way we dump information to our console. We're going to use it to send a message when we start up our Node app.

Here is our **`server.js`** file's contents.

```
console.log('Wow! A tiny Node app!');
```

Now we can start up our Node application by going into our command line and typing: `node server.js`

```
$ node server.js
```

We should see our message logged to the console. Remember that since Node is JavaScript on the server, the console will be on our server. This is in contrast with the client-side console that we'll find in our browser's dev tools.

Console Log

TIP: Restarting a Node Application on File Changes

By default, the `node server.js` command will start up our application, but it **won't restart when file changes are made**. This can become tedious when we

are developing since we will have to shut down and restart every time we make a change.

Luckily there is an npm package that will watch for file changes and restart our server when changes are detected. This package is called [nodemon](#) and to install it, just go into your command line and type:

```
$ npm install -g nodemon
```

The `-g` modifier means that this package will be installed globally for your system. Now, instead of using `node server.js`, we are able to use:

```
$ nodemon server.js
```

Feel free to go into your `server.js` file and make changes. Then watch the magic happen when application restarts itself!

We'll be using `nodemon` for the rest of this booklet.

Running an HTTP Server with Node

Code for this example found here: <https://github.com/scotch-io/node-booklet-code/tree/master/app2>

Node doesn't just have the ability to process JS files like we just did; it can also create an HTTP server. We're going to look at setting up an HTTP server with Express (a Node framework) to serve an HTML file.

In the first app, we only logged something to the console. Moving forward, we will take what we learned a step further so that we can serve a website to our users. We'll be another step closer to fully-fledged web applications.

Let's stick with the same application, add the Express framework, and deliver an HTML file. We'll need the same files (`package.json`, `server.js`), and we'll add a new `index.html` file.

Express: A Node Framework

One of the biggest strengths of Node is that it has support for many packages. The community submits many packages to [npm](#) and at the time of this writing, there are 311,863 packages with over 201,817,776 downloads in the last day. That's a lot of movement!

Packages extend the functionality of our application and there are packages for so many different use-cases. You may have heard of task-runners like [Grunt](#) and [Gulp](#) or even CSS processors like LESS can be packages.

Express is a lightweight platform for building web apps using NodeJS. It helps organize web apps on the server side. The [ExpressJS website](#) describes Express as "a minimal and flexible node.js

web application framework".

Express hides a lot of the inner workings of Node, which allows you to dive into your application code and get things up and running a lot faster. It's fairly easy to learn and still gives you a bit of flexibility with its structure. There is a reason it is currently the most popular framework for Node. Some of the big names using Express are:

- MySpace
- LinkedIn
- Klout
- Segment.io

For a full list of Express users, visit the [Express list](#).

Express comes with several great features that will add ease to your Node development.

- Router
- Handling Requests
- Application Settings
- Middleware

Don't worry if these terms are new to you, just know that Express makes Node development much easier and is a joy to work with.

Installing Express

The packages for a specific Node application are defined in its `package.json`. To get packages installed, you can employ one of two methods:

- **Method #1:** Write it into `package.json`
- **Method #2:** From the command line using `npm install`

We're going to use method #2 here. Go into your command line and type:

```
$ npm install express --save
```

The `--save` modifier will let npm know that it should write that package to your `package.json` file. If you run that command and go look in the `package.json` file, you'll notice it will be in a `dependencies` section of your file. You'll also notice that a new folder called `node_modules` is created. This is where Node stores the packages for a certain project.

This is what makes sharing projects between developers and collaborators very easy. Just send other users your project and they run `npm install` to install everything in the `dependencies` section.

Since we now have Node and Express ready, let's use both to create an HTTP server and serve up an HTML file to our users.

Creating an HTTP Server and Serving an HTML File

Let's get the easy part out of the way, the HTML file. In your project, create a new **index.html** file and place the following inside:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My Node App!</title>

  <!-- CSS -->
  <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.2
  <style>
    body { padding-top:50px; }
  </style>
</head>
<body class="container">

  <div class="jumbotron">
    <h1>My App!</h1>
  </div>

</body>
</html>
```

We'll be linking to the CSS framework [Bootstrap](#) through [Bootstrap CDN](#) to help us get quick CSS styling for this demo.

Let's move forward and create our HTTP server in Node using Express. Delete everything in your `server.js` file and here is what we will need:

```
// grab express
var express = require('express');

// create an express app
var app = express();

// create an express route for the home page
// http://localhost:8080/
app.get('/', function(req, res) {
  res.sendFile('index.html');
});

// start the server on port 8080
app.listen(8080);
// send a message
console.log('Server has started!');
```

That file is all that is required to use Express to start an HTTP server and send an HTML file!

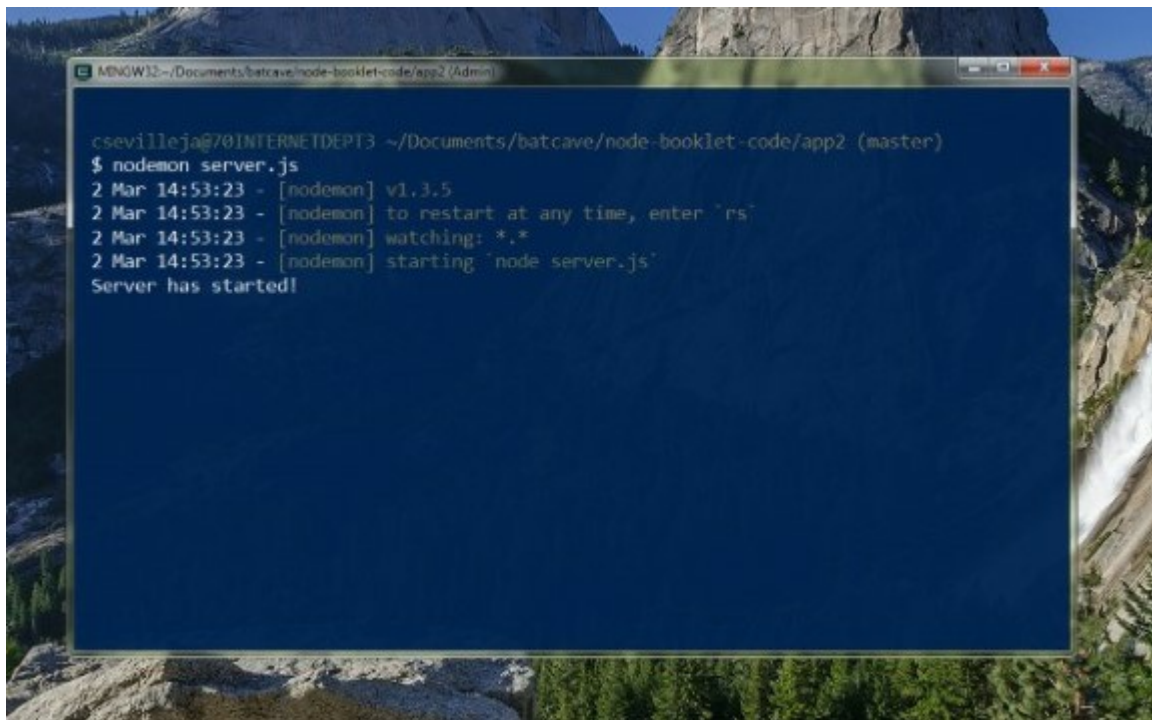
`require()` is the main way that we call packages in Node. Once we have created an Express application in `app`, we can define routes on it using the HTTP variable. `app.get()` creates a GET route for `/`.

When creating routes, we will always have access to the `req` (request) and `res` (response). The request contains information from the browser like HTTP agent, information passed in and more. The response is what we will send back to the user. We are using `sendFile()`, there are more things that can be done like sending back JSON data using `res.json()`.

We can start the server using `app.listen()` and passing in the port that we want 8080.

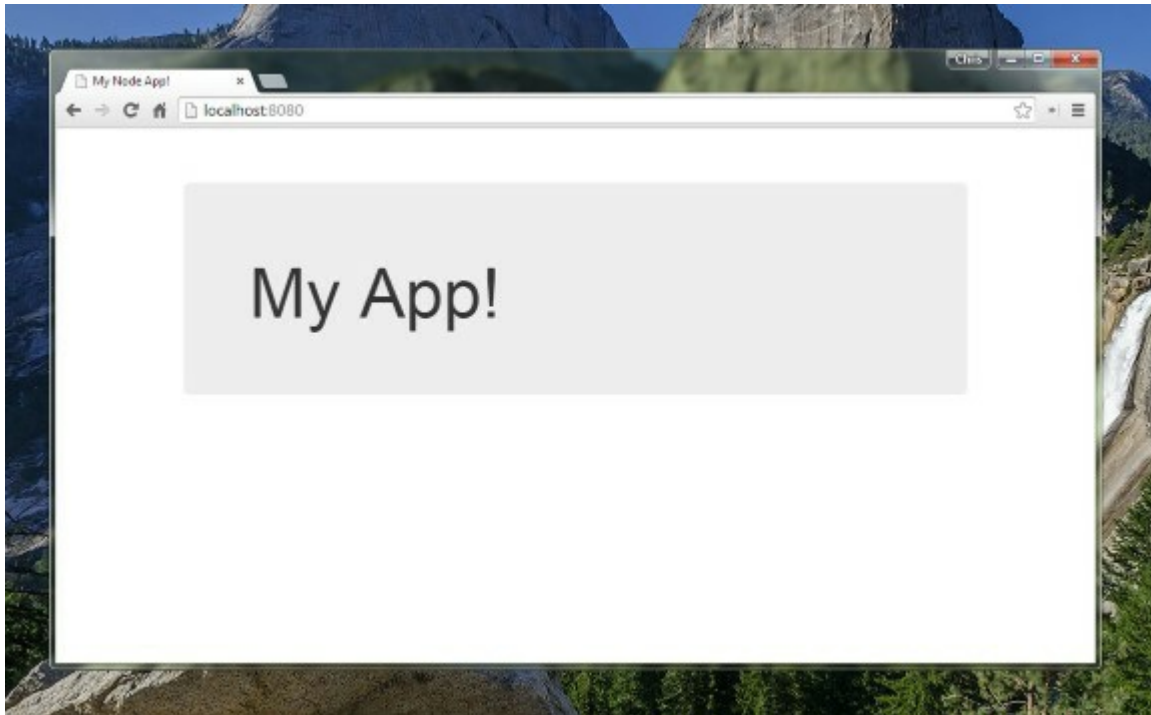
Let's make sure everything works by going into the command line to process this file and start our server.

```
$ nodemon server.js
```



```
MINGW32~/Documents/batcave/node-booklet-code/app2 (Admin)
csevilleja@70INTERNETDEPT3 ~/Documents/batcave/node-booklet-code/app2 (master)
$ nodemon server.js
2 Mar 14:53:23 - [nodemon] v1.3.5
2 Mar 14:53:23 - [nodemon] to restart at any time, enter `rs`
2 Mar 14:53:23 - [nodemon] watching: *.*
2 Mar 14:53:23 - [nodemon] starting `node server.js`
Server has started!
```

Then we can view our site in our browser at <http://localhost:8080>:



Whenever we start a server with Node, it will be placed at `http://localhost:PORT_NUMBER`.

This is a very easy and quick way to create an HTTP server and start developing. Node and Express can be used to build amazing applications or if you need it, it can just create a simple server to work on.

Great! We've done a lot with Node already:

- Installed Node
- Processed a very simple file
- Used npm to install a package
- Create an HTTP server with Express
- Serve an HTML file

Let's take this a step further and create an application that actually shows relevant data.

Our First Node App

Code for this example found here: <https://github.com/scotch-io/node-booklet-code/tree/master/app3>

For this Node application, we'll be building upon the concepts we've already learned in this booklet. A common task when building out any sort of application is to use third-party data. We will be connecting to the Instagram API and grabbing data from there to show our most recent pictures.

Requirements

- Use Express as the Node framework
- Use the Instagram Developer API
- Use the [instagram-node](#) package
- View our Instagram photos
- Template a Node app with [EJS](#) and the [EJS package](#)

Directory Structure

```
- public/  
  ----- css/  
  ----- style.css  
- views/  
  ----- pages/  
  ----- index.ejs  
  ----- partials/  
  ----- head.ejs  
  ----- header.ejs  
  ----- footer.ejs  
- package.json  
- server.js
```

We have the same structure to start our Node application. `package.json` and `server.js` are still there. We will be serving public files (CSS/JS/images) from the **public/** folder.

You'll notice that our views are separated into **partials/** and **pages/**. Partials will be reusable components like the header and footer of our site. It's good practice to separate these out so that we can keep our code [DRY](#).

[EJS](#) will be the templating engine and it is very commonly used within Node applications. This helps in a number of ways over having plain HTML files. We are able to:

- Display dynamic data sent from the server
- Repeat over variables and lists
- Template our applications

Let's work with the Instagram data in Node first in our `server.js` file. Once we have the Instagram data that we need, we'll move over to our view files to display it.

Setting Up Our Application

Let's create a new folder for this application. It will be a good practice to start from scratch so we can get used to making Node applications.

package.json

Once your new folder is created, jump into the command line and start your Node application

using:

```
$ npm init
```

Fill in your information however you like and then we'll install the packages that we need.

```
$ npm install express ejs instagram-node --save
```

This will install these three packages into the **node_modules/** folder and add them to the dependencies section of `package.json`.

server.js

We will now start up our `server.js` file. The main things we need to do in this file are:

- Grab the packages we need (Express, EJS, instagram-node)
- Configure these packages
 - Set Instagram API key
 - Set EJS as our templating engine
 - Set Express assets directory (for CSS)
- Create a route for the home page
 - Grab our Instagram images
 - Pass to our views
- Start the server

```
// GRAB THE PACKAGES/VARIABLES WE NEED
// =====
var express = require('express');
var app     = express();
var ig      = require('instagram-node').instagram();

// CONFIGURE THE APP
// =====
// tell node where to look for site resources
app.use(express.static(__dirname + '/public'));

// set the view engine to ejs
app.set('view engine', 'ejs');

// configure instagram app with client-id
// we'll get to this soon

// SET THE ROUTES
// =====
// home page route - our profile's images
app.get('/', function(req, res) {

    // use the instagram package to get our profile's media
    // render the home page and pass in the our profile's images
    res.render('pages/index');

});
```

```
// START THE SERVER
// =====
app.listen(8080);
console.log('App started! Look at http://localhost:8080');
```

We have grabbed our packages, set the configurations that we need to, created one home page route, and started our server. This can be done quickly in Node since we have the ability to `app.use()` and `app.set()` on our Express application.

These configurations can be found on the Express and `ejs` package pages and usually packages will provide very clear instruction on their GitHub repository or npm page.

While we used `res.sendFile()` earlier, `ejs` provides the `res.render()` function. By default, Express and `ejs` will look in a `views/` folder so we don't have to specify `views/pages/index`. `pages/index` will be enough.

We will also grab and create an `ig` object using `require('instagram-node').instagram()`. You can find these instructions on the [instagram-node](#) npm page.

View Files

Before we can test this server to make sure that everything is working, we'll need a view file to show! We're going to move quickly through the view files since these aren't the main focus of our Node application.

views/partials/head.ejs

```
<meta charset="UTF-8">
<title>My First Node App!</title>

<!-- CSS -->
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootswatch/3.3.2/co.
<link rel="stylesheet" href="css/style.css">
```

We're grabbing [Bootstrap](#) from a [CDN](#) for quick styling. We picked one of the Bootstrap files from the [Bootswatch](#) section of the site to switch up from the default Bootstrap styling.

We are also loading a stylesheet `css/style.css`. It is in our **public/css/** folder, and since we set `express.static()` in `server.js`, our application will serve assets from the **public/** folder.

views/partials/header.ejs

```
<nav class="navbar navbar-inverse">
<div class="container-fluid">
  <div class="navbar-header">
    <a href="/" class="navbar-brand">
      <span class="glyphicon glyphicon-picture"></span>
      My Instagram Photos
    </a>
  </div>
</div>
```

```
</nav>
```

Nothing too crazy here. Just a Bootstrap navbar with a link back to the home page. The span is one of the Bootstrap [glyphicons](#).

views/partials/footer.ejs

```
<p class="text-center text-muted">
  Copyright &copy; 1800-2050 &middot; The Coolest App in the World
</p>
```

Good old copyright all the way back from the 1800s to the future!

views/pages/index.ejs

```
<!DOCTYPE html>
<html lang="en">
<head>
  <% include ../partials/head %>
</head>
<body class="container">

  <header>
    <% include ../partials/header %>
  </header>

  <main>
    instagram photos will go here
  </main>

  <footer>
    <% include ../partials/footer %>
  </footer>

</body>
</html>
```

We are using `include` to pull in the partials. This helps our application grab the other ejs files. In ejs, the `<%` and `%>` tags will be how we display information.

public/css/style.css

```
body {
  padding-top: 50px;
}

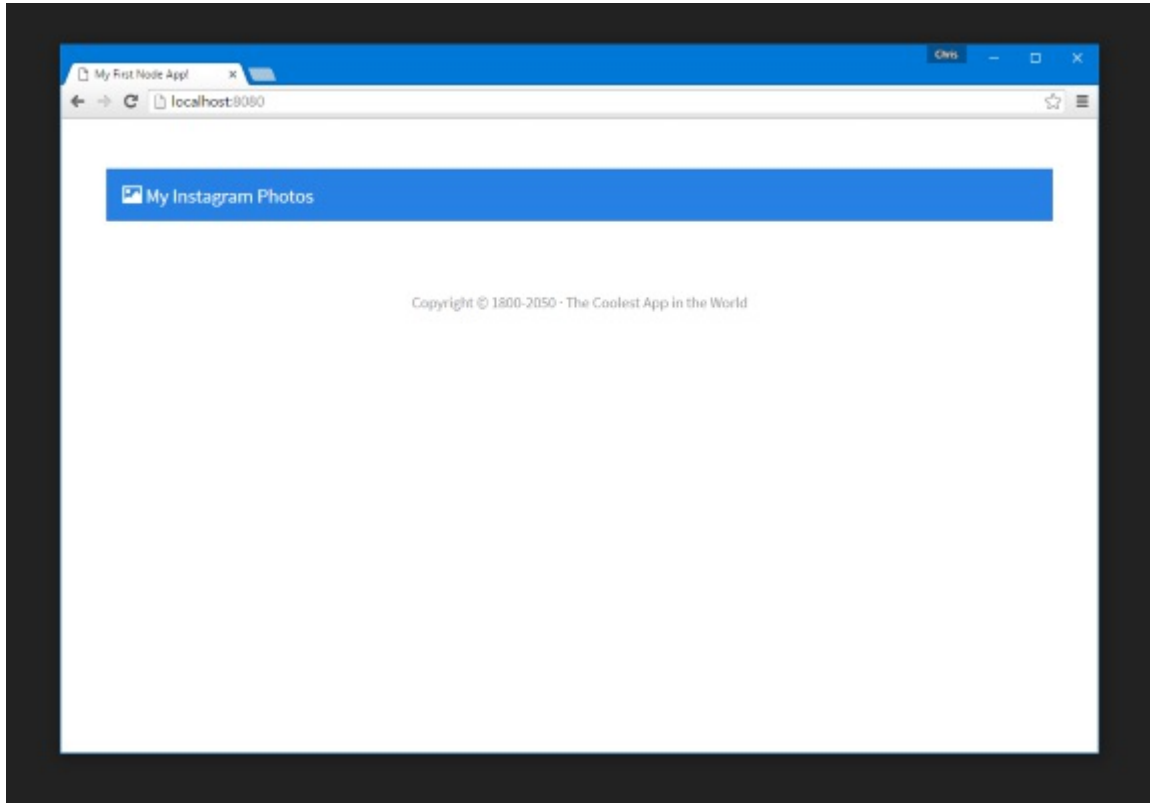
footer {
  padding: 50px;
}
```

This is all we'll add for styling right now. We'll style our images after we get them into our application.

With all that out of the way, let's start our server and view what we've created in our browser:

```
$ nodemon server.js
```

We'll be able to see the site in browser at <http://localhost:8080>.



Next, we'll use the `instagram-node` package to get popular photos and display that in our app.

The Instagram API

In order to use the Instagram API, we have to register for free as Instagram developers. The main things we will need for this application are a `client_id` and `client_secret`. We can grab these after we create our account and register a new application.

These credentials are necessary for accessing Instagram's resources including an access token. An access token as the name goes is a string based token that Instagram uses to authenticate a user trying to access an API endpoint. The `instagram-node` SDK is capable of exchanging `client_id` and `client_secret` for an `access_token` but that is beyond the scope of this book so we just use a third-party service, [PixelUnion](#).

You can receive a `client_id` and `client_secret` from the [Instagram Developer](#) site if you need to.

Instagram Manage Clients Log In

Search Documentation

- Overview
- Authentication
- Restrict API Requests
- Real-time
- Mobile Sharing
- API Console
- Endpoints
- Links
- Embedding
- Libraries
- Support
- Platform Developers

Hello Developers.

The first version of the Instagram API is an exciting step forward towards making it easier for users to have open access to their data. We created it so that you can surface the amazing content Instagram users share every second, in fun and innovative ways.

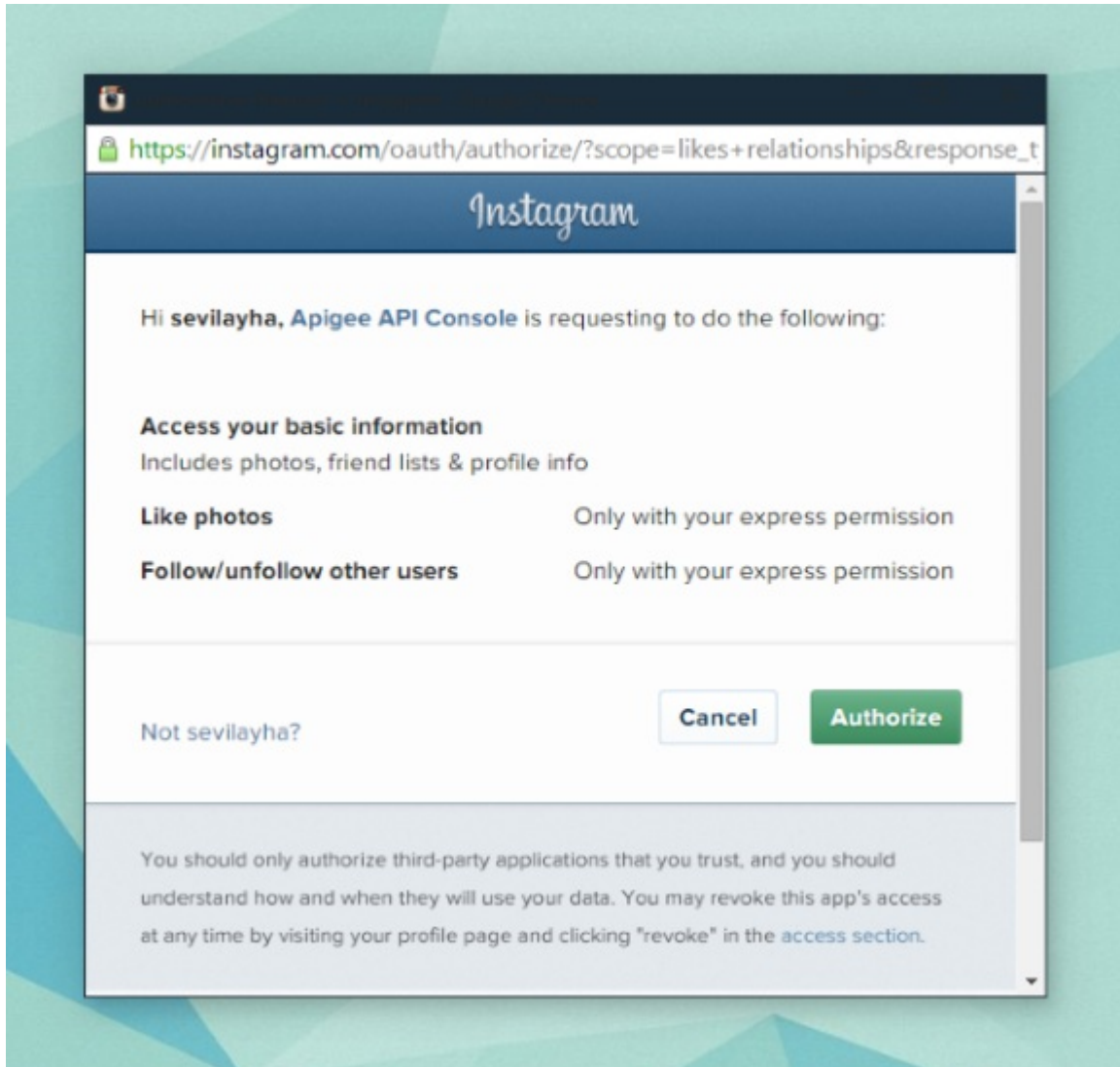
Build something great.

[Register Your Application](#) then [dive into the documentation](#)

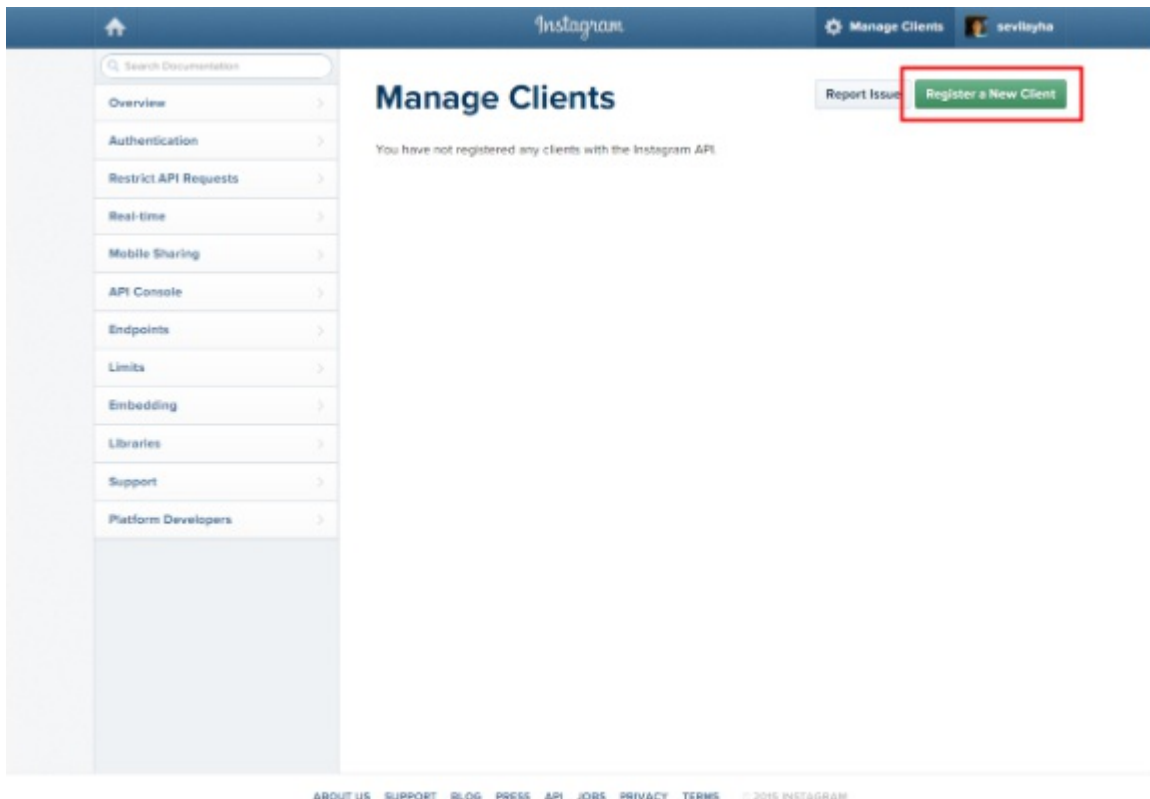
Getting Started

- 1 Register**
We'll assign an OAuth client_id and client_secret for each of your applications.
- 2 Authenticate**
Have our user authenticate and authorize your application with Instagram.
- 3 Start making requests!**
Make requests to our API Endpoints with your authenticated OAuth credentials.

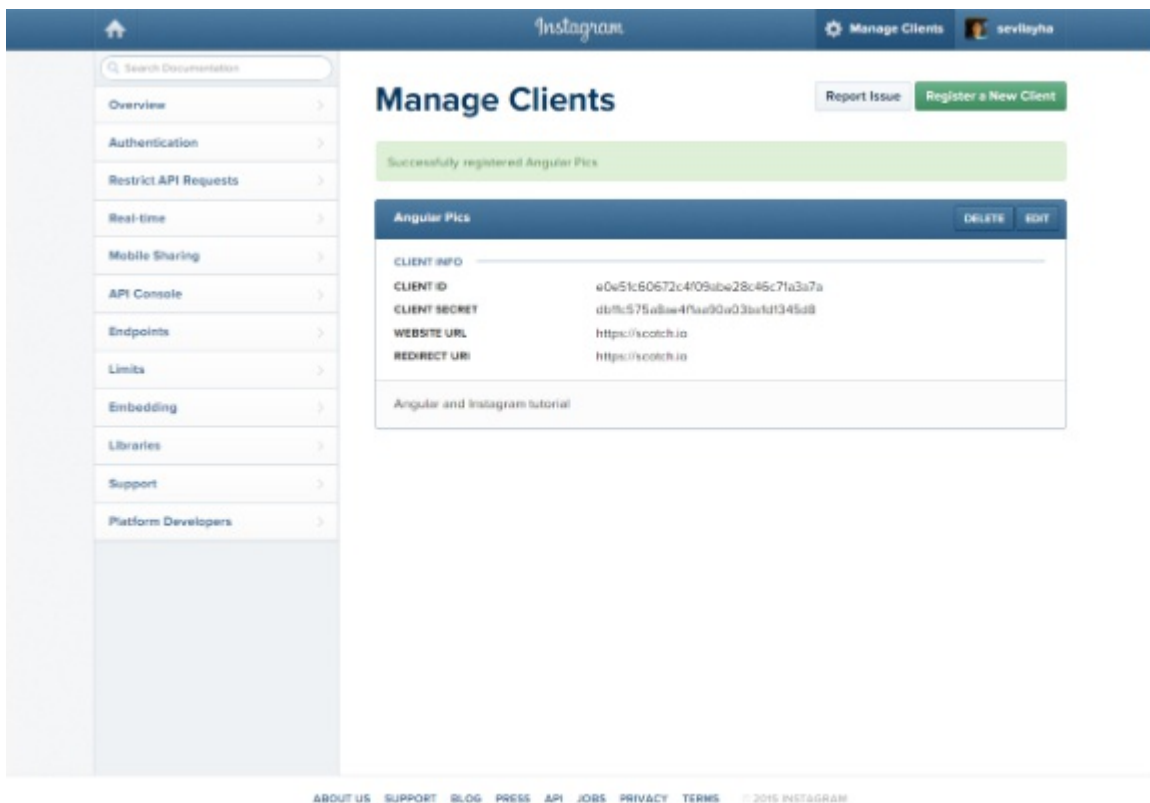
Go ahead and login.



Then we will go on to create our new client (or application).



After we go through and create our client, we'll be given a **client_id** and **client_secret**.

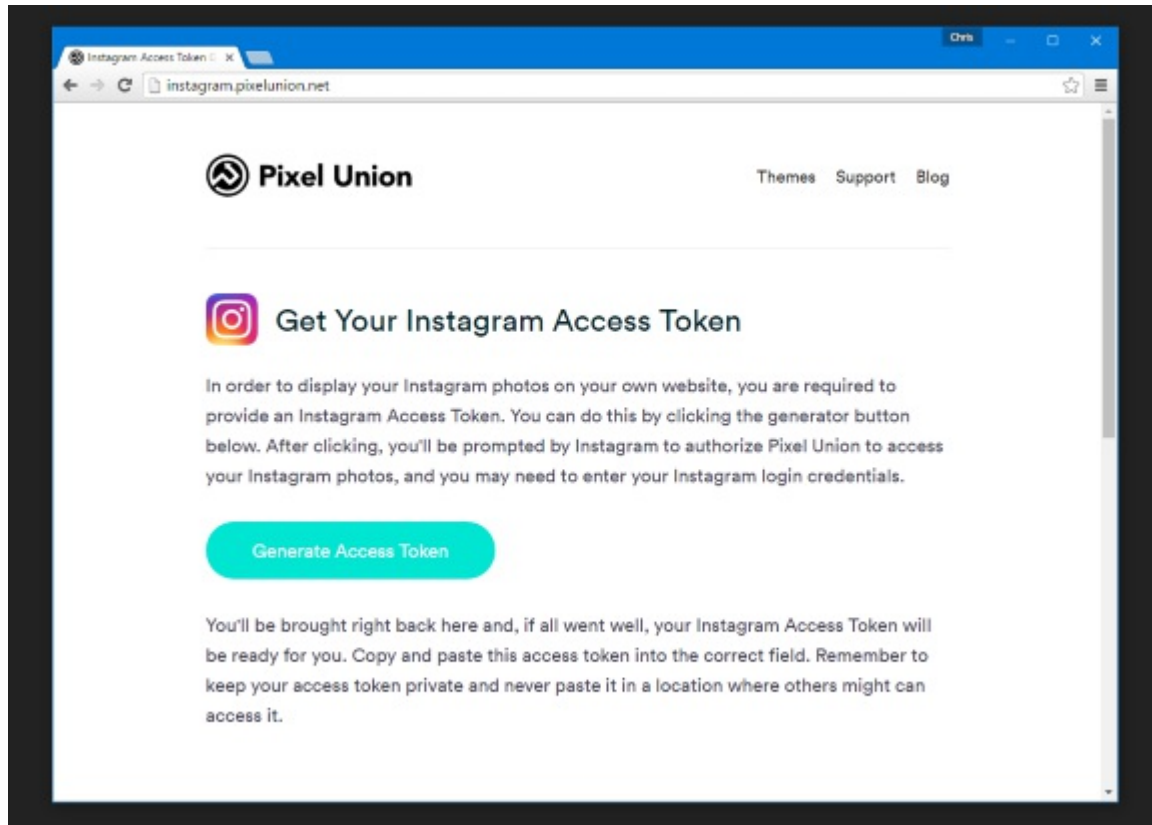


This is exactly what we'll need to gain access to the Instagram API through our Node application. We can use the `CLIENT_ID` and `CLIENT_SECRET` to authenticate our apps and access photos.

Getting an Access Token

Another way we can authenticate our applications is with an access token. This is the equivalent to when you see an application say **Authenticate with Instagram** or **Login with Facebook**. We are using OAuth2 to get an access token by logging in.

We can get an access token very simply using [<http://instagram.pixelunion.net/>].



Keep your new `access_token` safe as this can provide access to your Instagram account as we'll soon see.

Now that we have the credentials necessary to use the Instagram API, let's take a tour of the API.

Exploring the Instagram API

Let's take a second to look through the API to see what information we can grab. Often, companies will provide an API explorer so that you can use a convenient interface to see the JSON data that will come out of an API. A couple big companies that offer API explorers are [Facebook](#) and [Twitter](#).

Instagram also offers an API explorer through a service called [Apigee](#).

We can access the explorer from the [Instagram dashboard](#) and fullscreen from Apigee:

[Apigee Instagram API Explorer](#)

Once you open up the API explorer, it can be a little daunting at first. The left panel will be where we can see all the API calls. The one we want to focus on is the `GET users/self/media/recent` call. This is a variation of the `GET users/{user_id}/media/recent`.

Just replace `{user_id}` with `{self}`. This will make sure we get our users photos.

Select an API method

Users

-  GET users/{user-id} 
-  GET users/self/feed 
-  GET users/{user-id}/media/recent 
-  GET users/self/media/liked 
-  GET users/search 

Relationships

-  GET users/{user-id}/follows 
-  GET users/{user-id}/followed-by 
-  GET users/self/requested-by 
-  GET users/{user-id}/relationship (GET) 
-  POST users/{user-id}/relationship (POST) 

Media

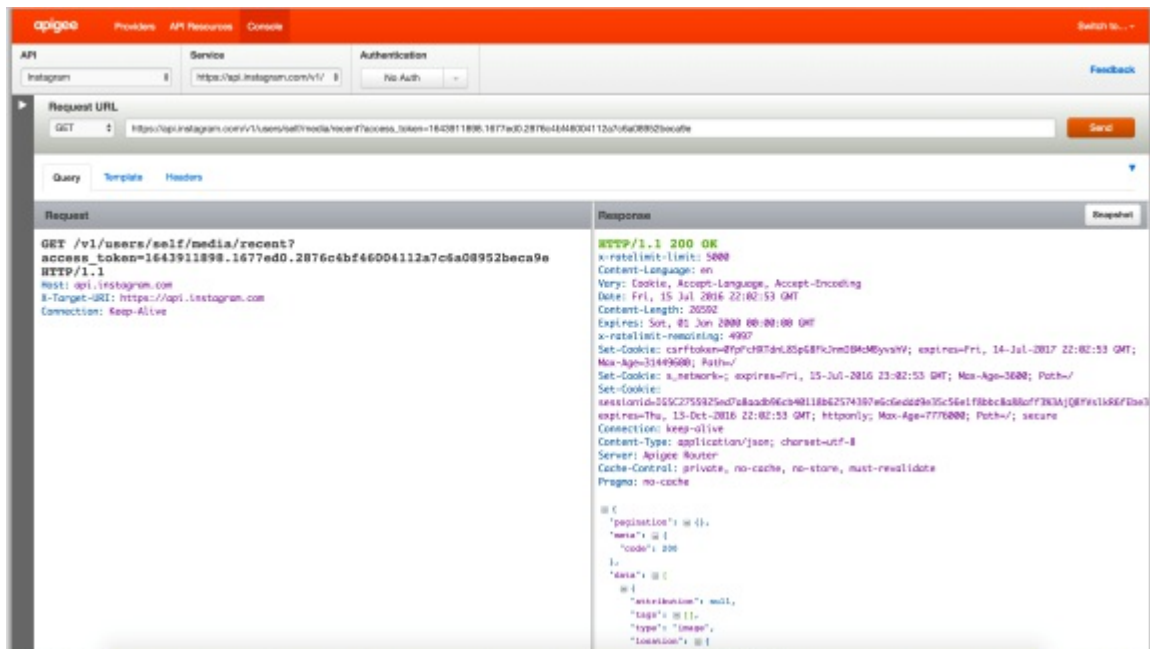
-  GET media/{id} 
-  GET media/search 
-  GET media/popular 

Comments

Let's add our `access_token` credentials to the URL like so:

`https://api.instagram.com/v1/users/self/media/recent?access_token=[ACCESS_TOKEN]`

That is accessing the API with the access token we grabbed earlier. We can alternatively click on **Authentication** and select **OAuth2**.



On closer inspection, we can see we get an array of the images called `data`.

Response

1

```
Pragma: no-cache
Date: Thu, 05 Feb 2015 19:34:50 GMT
Vary: Cookie, Accept-Language, Accept-Encoding
X-RateLimit-Remaining: 4998
Content-Type: application/json; charset=utf-8
```

```
{
  "meta": {
    "code": 200
  },
  "data": [
    {},
    {},
    {},
    {},
    {},
    {},
    {},
    {},
    {},
    {},
    {},
    {}
  ],
  "attribution": null,
  "videos": {
```

If we take a look at a single object in the data array, we can see that it is one photo with all the information that we need.

```

{
  "meta": {},
  "data": [
    {
      "attribution": null,
      "videos": {},
      "tags": [],
      "location": null,
      "comments": {},
      "filter": "Normal",
      "created_time": "1425339261",
      "link": "https://instagram.com/p/zvnISdy5_h/",
      "likes": {},
      "images": {},
      "users_in_photo": [],
      "caption": {},
      "type": "video",
      "id": "932135741285965793_218374838",
      "user": {}
    },
    {},
    {},
    {},
    {},
    {},
    {},
    {}
  ]
}

```

We can grab the `comments`, `likes`, `images`, and `user` info. We're going to use these when building out our application. We have what we need here:

- `client_id`
- `client_secret`
- `access_token`
- An understanding of the API data

Let's move forward and get this data into our application and showing to our users.

Instagram API Terms and Rules

I'd encourage you to always read the [API terms and conditions](#) before using it. There are stories of applications that have grown successful only to have their API access yanked away from them because they violated the terms.

The main points in Instagram's terms are:

- Cannot replace or replicate `instagram.com` or `instagram` apps
- Can't show more than 30 at a time
- Can't participate in any "like", "share", "comment", or "follower" exchange programs

Most of these rules are pretty standard when using APIs. The creator of the API wants to ensure that its users aren't being bombarded with ads, spam, or fake users. This helps keep the community at a higher level of conduct.

Well if we can't replicate the website or its applications, what's the point in working with their API? That is a good question. While Instagram seems to be pretty lenient on that rule since there are sites and apps that pretty closely replicate the core apps features, there are also very neat applications like [printstagram](#), that will allow you to print out square sized images of your grams.

Instagram-Node

Back in our application, we will need to configure the `instagram-node`. Let's add this line in our configuration section of our `server.js` file:

```
// CONFIGURE THE APP
// =====

...

// configure instagram app with client_id, client_secret, and access_token
ig.use({
  // get access token here: http://instagram.pixelunion.net/
  access_token: 'MY_ACCESS_TOKEN',
});

// alternatively we can use the client_id and client_secret
// for now we'll use the access_token way
// ig.use({
  // get these from when we create our app as an instagram developer
  // https://www.instagram.com/developer/
  // client_id: 'MY_CLIENT_ID',
  // client_secret: 'MY_CLIENT_SECRET'
// });

...
```

We've filled in our client credentials and personal access token.

In the future, when you want to build out your larger application, you can authenticate your users and use their personal `access_token` to display their information.

The `instagram-node` package will handle all the calls to the API for us. We don't have to worry about appending our `access_token` to any URL like we did in the API explorer.

The package also wraps a lot of the API calls to make them easier to use. The list of the API

calls can be found on their [docs page](#). The call that we want is for our recent media. That call is:

```
ig.user_self_media_recent(function(err, medias, pagination, remaining, lim
```

We'll be using this call to pass the `medias` to our view.

Getting the Data to Our HTML Views

Express let's us pass data to our views through our routes by passing in an object of data.

Sample Data to EJS

For example, we can pass in a simple message to our view by using:

```
app.get('/', function(req, res) {
  res.render('pages/index', { message: 'I am data!' });
});
```

Using `ejs` in our view, we would be able to display the `message` variable by using:

```
<% message %>
```

Using Instagram Data

Let's use these concepts with the `instagram-node` call to grab popular media. In our `server.js` file, we will be returning the popular media to the home page. Since this is the case, this call will be made in our `app.get('/'... route`.

Grab Instagram Data in Node

In `server.js` where we defined our main home page route, we will replace that entire route with:

```
// home page route - popular images
app.get('/', function(req, res) {

  // use the instagram package to get popular media
  ig.user_self_media_recent(function(err, medias, pagination, remaining,
    // render the home page and pass in the popular images
    res.render('pages/index', { grams: medias });
  });

});
```

When we use the `ig.user_self_media_recent()` call, we gain access to the `medias` object. This contains all of the `data` that we saw in the API explorer.

Using `res.render()`, we can pass all the `medias` data to our view as an object called `grams`. Let's use this object in our view now.

Display Instagram Data in View

Inside of the `index.ejs` file, we will loop over this `grams` object. We will display the:

- Instagram picture
- Number of likes
- Number of comments

Using `ejs`, we can loop over the object with `grams.forEach()`. Replace what is in the `<main>` section of our site with the following:

```
...  
  
<main>  
<div class="row">  
<% grams.forEach(function(gram) { %>  
<div class="instagram-pic col-sm-3">  
  
    <a href="<%= gram.link %>" target="_blank">  
          
  
    <div class="instagram-bar">  
        <div class="likes">  
            <span class="glyphicon glyphicon-heart"></span>  
            <%= gram.likes.count %>  
        </div>  
  
        <div class="comments">  
            <span class="glyphicon glyphicon-comment"></span>  
            <%= gram.comments.count %>  
        </div>  
    </div>  
  
</div>  
<% }); %>  
</div>  
</main>  
  
...
```

We have created a link to the image on Instagram. We also have an `instagram-bar` to display the number of likes and comments.

By looping over the `grams` object, we have access to everything that was returned by the API. You can use the API explorer to look through the data and call the necessary items.

For the image, we are using `images.standard_resolution.url` and for likes and comments respectively, we can use `likes.count` and `comments.count`.

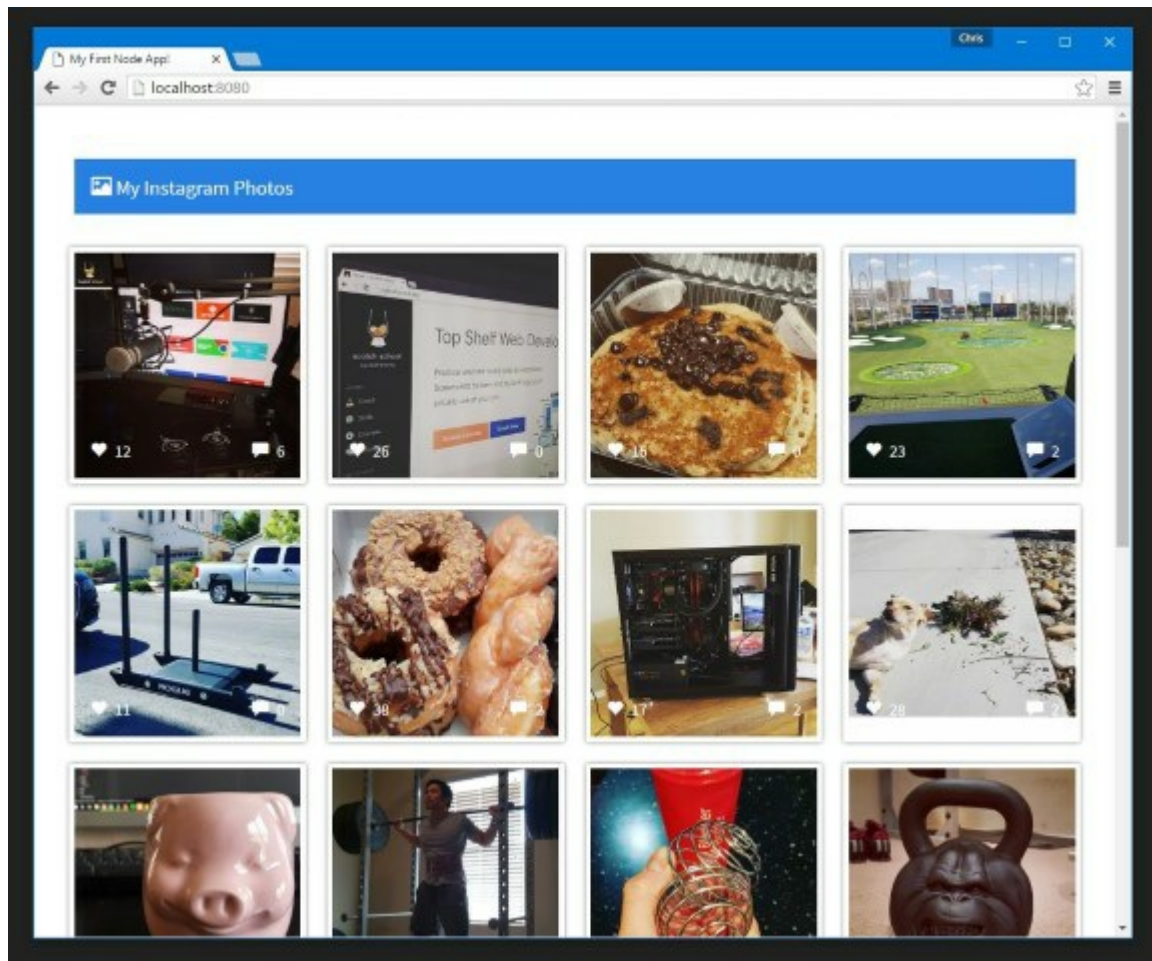
If we wanted information about the user, we could use `user.username` and `user.profile_picture`. The caption can be grabbed using `caption.text`.

Last step is to style up our images. Add the following to your `style.css` file:

```
.instagram-pic {
  position:relative;
  padding:10px;
  overflow:hidden;
}
.instagram-pic img {
  padding:5px;
  border-radius:2px;
  box-shadow:0 0 5px rgba(0, 0, 0, 0.5);
}
.instagram-bar {
  position:absolute;
  bottom:30px;
  width:100%;
  left:0;
  padding-left:30px;
  padding-right:30px;
  color:#FFF;
}

.instagram-bar span {
  margin-right:5px;
}
.instagram-bar .likes {
  float:left;
}
.instagram-bar .comments {
  float:right;
}
```

Finally we have our app with the popular Instagram images!



Conclusion

To recap, we've learned a bit about how Node works and why it's so neat. We've also been able to:

- Process a JS file with Node
- Install packages using npm
- Create an HTTP server with Node and Express
- Used a package to grab API data
- Templated an application with EJS
- Displayed data with EJS

The concepts here can apply to many different types of applications. Play around with the many packages available on npm. The possibilities of what can be done with Node are endless.

You can pair it with a frontend framework like [Angular](#) and a database system like [MongoDB](#) to create a MEAN Stack application. For more information on that, check out [Scotch School](#) where we'll be covering full video courses on popular topics like Node, JavaScript, Angular, and more.

For more Node and JavaScript tutorials, be sure to check out [scotch.io](#) as we update weekly

with new articles and video courses.