Web application development

(Or how to get a job ASAP!)

PROF. GABRIEL CASTILLO

Backend development

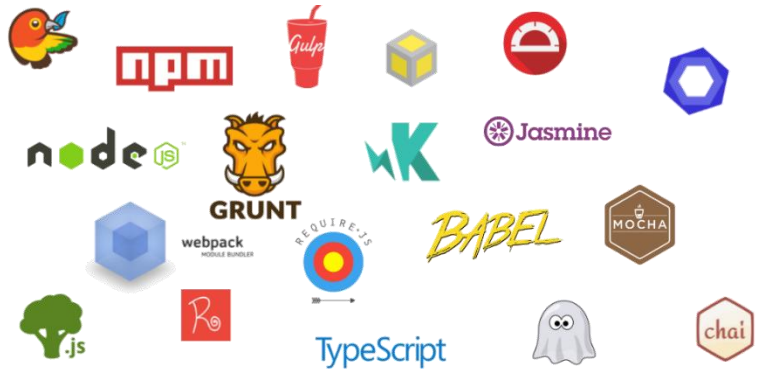
Introduction

What is backend development?

It is the part of a system that allows us to save and retrieve information and introduce business logic into our system.

This is where the components that make up the secret sauce of your application will be stored. Here, you will retrieve the information you have gathered, process it, slice it, dice it and then recreate it in such a way that you will then be able to display it in the front end for the user to use.

Opposed to the front end, in the backend we will be using different technologies than the ones used in the front-end.



Introduction

Over the time several technologies have been developed in order to fulfill this task. Starting with Perl in the early days of the internet, nowadays we have a plethora of options such as

php + CakePhp

Ruby on Rails

Asp.net

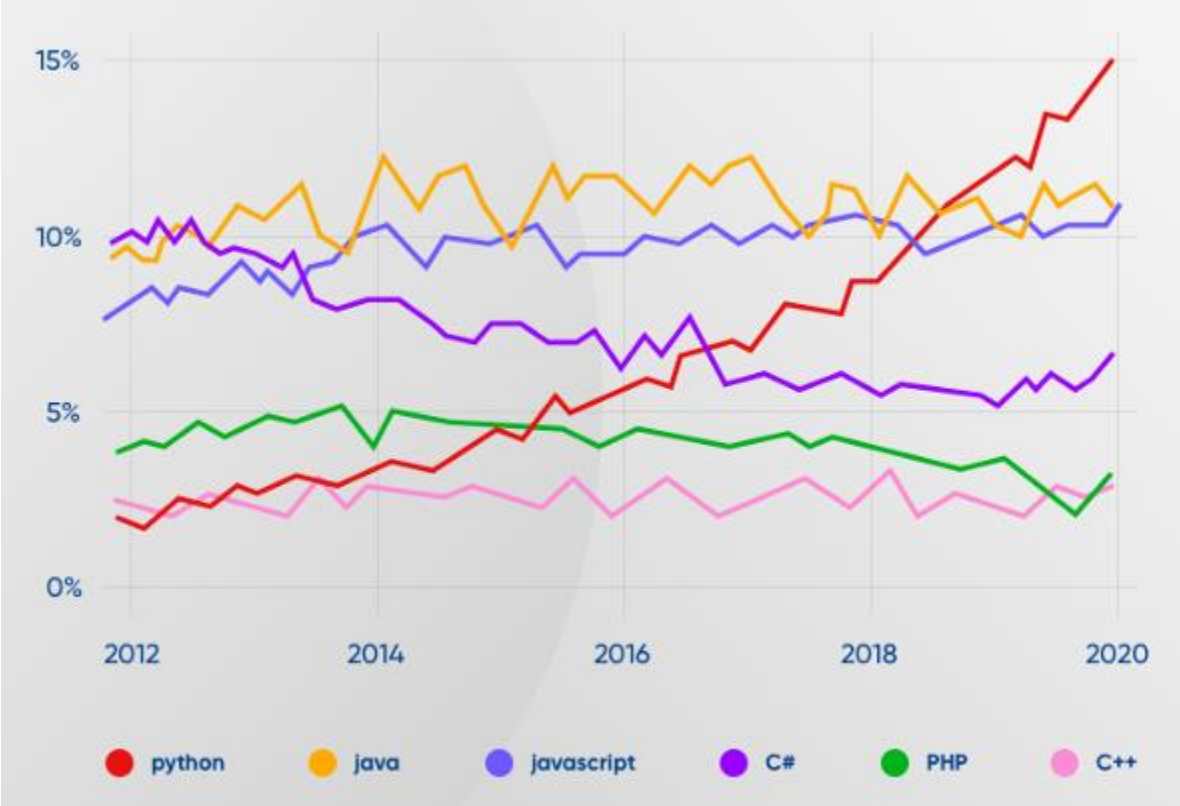
Java + Spring

Python + django

Node.js + express

Introduction

Usage trending for different backend technologies.



UNIVERSIDAD PANAMERICANA

Node.js

Introduction

We will be working with Node.js because it is a backend technology based on Javascript, thus we will not have to learn something new to use it.

You will have to be careful when you are developing, since sometimes it is easy to confuse what part are you working on; thus causing some leaking of code into places that it does not belong.

But why do we have to do introduce something different such as Node.js? What is its difference with vanilla JS?

- JS is restricted to work within the reaches of the browser. As such, it can not access the file system or any of the computer resources.
- The reason is that since it is very easy to run and load through the browser, anybody could then lure into your system without you even knowing!!
- In the past, some hackers have managed to place workarounds and collected information from users exploiting the locks on JS. Nowadays JS is a good enough secure technology.

What is Node.js

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications.

The main paradigm it uses is an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Node.js runs outside of the browsers' environment, thus giving it full access to the resources of a machine!

This enables Node.js to build desktop applications as well as web applications.

- E.g. Atom is built on top of Node.js

Our first app

Let’s practice our command line

- Open your command terminal (git bash, hyper, terminus, putty, VS Code terminal, etc)
- Navigate to the desired location you want to place things in.
  - If you still don’t know how to navigate with command line, please refer to the Command line Lab in the website

Let’s test if our installation was successful by running the **node --version** command

- You should see the version of your node installation, if this doesn’t work. Check your installation process
- Once there, create a new directory called node-test: > **mkdir node-test**
- Navigate inside that folder: > **cd node-test**
- Create a new file called index.js: > **touch index.js**
- Edit the file and add the line: > **console.log(“hello world”);**

Before, this code was only able to run in our console, but now we’ll see how with the node command we can run it locally!! > **node index.js**

Node.js REPL

REPL stands for Read Evaluation Print Loop.

This is just the command line interface we can use when working with Node.js

To access it we just type **node** in the command line, and the prompt will change to a **>** sign

Once inside it will work exactly as our console worked in chrome. We can create variables, define functions, call them, etc.

To finish the REPL execution you can either type .exit or press Ctrl+C twice.

- We will be using the Ctrl+C method more often, since sometimes when a Node server is running, we can not get a hold of the terminal anymore.

Node.js native modules

Node.js comes preloaded with a bunch of modules that will allow you to start working right away with multiple things, such as manipulating the filesystem, do encryption, listen to web ports, manage events, threads, error handling, etc.

- The list of all the native modules is available in the documentation link in the resources of the class website.

To use any of the native modules inside of Node.js, we will have to create a constant to keep its reference when in use. Thus, if we want to use the module, it would look like this:

- E.g. Using the filesystem module to read a file

```
const fs = require('fs');
fs.open('path/to/file.txt', 'r', (err, fd) => { <DO something with the file> });
```

Notice the use of the **const** keyword instead of **var** to create the **fs** reference constant.

Also see how we use the **require( )** function in order to load the module into the constant.

Node.js native modules

Once that code is placed in our .js file, we can just run it invoking its execution with the node command as previously explained.

There are a LOT of native modules that allows you to have full control over an application, so make sure that you at least surf the documentation so you can see how much you can do with it.

Node.js external modules

To use external modules, we will need a package manager. This will make sure that the packages we are using are updated and are safe to use.

NPM is the absolute standard in the Node.js environment to take on this task.

The good news is that NPM already comes bundled with Node.js, thus we don't need to do anything more to use it, just learn some basics.

To use these packages, we have to let Node know about them, so we will need to initialize our project to be able to hold the reference to those packages.

To do so, inside our project root folder we will execute the command: **npm init**

When you run it you will be prompted for various information.

Once you answer all the prompts a new file called ***package.json*** will appear in your project. This is a key file in a Node project, since it will keep track of the required modules needed for a proper execution of our project.

npm init

When you run the **npm init** command, you will be prompted for the following information:

- **Package name**: This will be the name of the module YOU are creating, or the name of your application. You can change it later, but this is how you will be able to launch your app.
- **Version**: the current version of your app. This can also be updated as your app evolves.
- **Description**: A summary of your app/module.
- **Entry point**: This is the main file of your application. The convention is that the file is called index.js
- **Test command** (Optional): We will discuss this if we are able to get to that topic
- **Git repository** (Optional): The path to the repository your project is int.
- **Keywords** (Optional): Keywords so it is easier to look for if you upload your project to npm index
- **Author** (Optional): The name of the creator. It is highly recommended that you DO put your name.
- **License**: The type of license under which you are releasing your code (ISC, CC, MIT, BSDx, etc)

Once your *package.json* file is in place, your app is ready to use external modules.

Installing external modules

Using them is very easy, first you have to identify what are the packages that you will require. The npm webpage (link in the reference section) is the best place to start looking.

Once identified, you just run the command: **npm install <package_name>**

- It will connect to the source and download the required files so you can start using that module
- Make sure the spelling of the package_name is exact to avoid confusions.

After you install it, you will notice a new entry in the *package.json* file named ***dependencies***.

- It will state the ***name of the packages*** and its ***versions***. This is very useful to control what *versions* of every package you need and will help *npm* keep your code up to date when newer *versions* of your used modules are available.

If needed, you can specify which version of the module you need, using the **@** character followed by the specific version you want.

e.g. **npm install dependency@1.1.3**

Installing external modules

You will also notice that in your *root* folder, you will nave a new folder called ***node_modules***. Inside of it, you will be able to see all the code of the installed modules you will need.

If you dig into the *node_modules* folder, you might notice that there are way more projects than just the one you installed. The reason is that the projects that your app depends on, might have their own dependencies, thus creating a recursive effect that will render your project with tons of seemingly useless modules.

Now you can use the modules just as native ones.

Updating external modules

Once you have your application running, you might realize sometimes that there are updates to some of the packages you are using.

To update, you have to run the command: **npm update <package_name>***

In order to update them, there are several ways to do so depending on the situation.

> npm update: Will update all the npm packages of the project

> npm update –g: Will update all the global npm packages of the installation

> npm update <package_name>: Will update ONLY the package specified

In your *package.json* file, you can specify some modifiers to control which versions your app will be able to manage when you run update:

- **Caret dependencies(^):** Approximately equivalent to version
- **Tilde dependencies (~):** Compatible with version

Express

Introduction

Express is a Node.js framework geared towards web applications.

Just as jQuery did for JS, Express uses basic Node.js code to ease web development.

Express is currently the #1 framework used by backend developers

Although at first the syntax and usage might seem complicated, once mastered express allows you to build web applications from the ground up in no time.

Creating an express server

Creating a new server is as simple as what we did with a Node.js application.

1. Create a new folder where your app will reside
2. Create the entry point in the file server.js
3. Initialize npm in that folder
4. Install the express package using npm: **npm install express**

Once express is installed, we can use a boiler plate code inside server.js, it looks like this:

```
1  const express = require("express");
2  const app = express();
3  app.get('/', (req, res) => {
4      res.send('Hello World!');
5  });
6
7  app.listen(3000, ()=>{
8      console.log("Example app listening on port 3000");
9  });
```

Dissecting the boilerplate

1. We import the express module.
2. We create an express server and store it in the app variable. This is a convention, so regularly you will see this app variable.
3. We configure our server to GET all the requests with the ***root*** "/" path. Once we receive a request, we will respond sending the *'Hello world'* message in the respond object.
   - Notice how that is taken care of in the callback anonymous function
7. We configure our server to listen to port 3000 and log a message when it starts.

```
1  const express = require("express");
2  const app = express();
3  app.get('/', (req, res) => {
4      res.send('Hello World!');
5  });
6
7  app.listen(3000, ()=>{
8      console.log("Example app listening on port 3000");
9  });
```

Accessing the server

Once our code is ready in *server.js*, we can run our code with the command: **node server.js**

In the command line we should see the message that the server is running.

If you access your server from a browser accessing port 3000 (localhost:3000) then you should see the “Hello world” message.

Notice how in the command line, the prompt seems to be hanging. To finish the server execution, you will have to press **Ctrl+C** to stop the server.

app.get

It is worth the effort further understand what does the get method does and how does it do it.

app.get(“<route_path>”, (req, res) => { <callback_function_body> });

The ***route_path*** will be the path after the root url that this get will be listening to. This is called the **route**.

- Let’s remember that the root url is the domain name of your app.
- The root of your application will be identified by the “/”

For the *callback_function* we use an *anonymous arrow function* that receives two parameters: **req** and **res**

- *For the parameters of callback functions in multiple methods, please refer to the documentation.*

app.get

The **req** parameter represents the ***request***. If you *console.log* it you will be able to see all the information contained here. This is the information we get from the client in the backend.

The **res** parameter represents the ***response***. This is the object that we will build with the updated information to return to the client.

The response method can hold anything we want. It can hold plain text, html, JSON objects, images, audio, stream packages, etc. If it can go in a regular HTML response, it can go in a response object.

The *req* and *res* names of the variables is a convention, these are not THE names of those parameters, you can find it in the documentation with different names.

Multiple routes

As we mentioned earlier, the get method registers the route that will be answered by every get method. Thus with that, we can divert and change the behavior of our server, to respond different things depending on what route gets hit by the browser.

These routes are key to the development of a full size application, since these are going to become the entry point for any request coming from the outside. An app can have as many routes as you want!

If the client tries to hit a route and there is no hit, then it will return a 500 error.

```
app.get("/", … );

app.get("/contact", …);

app.get("/directory/board", …);
```

Some utilities

There is a module that works with express, that is very useful to avoid all that stopping/starting of the server by hand. It is called **nodemon**

To install *nodemon* just do: **npm install –g nodemon**

This will monitor the code in search for changes, and once one is detected, it will automatically will restart and thus you will just need to start your server once and *nodemon* will deal with the refreshing of the server as well as the logging.

To run your server with *nodemon* just do: **nodemon server.js**

Some utilities

Another module that will prove to be very useful is the **body-parser** module. This module will allow us to receive requests and parse the parameters in a very simple way.

This used to be an independent module, but given how useful it has become, it is now part of the core modules of express.

In order to use it, we will just need to configure it. To do so, we add these lines:

```
app.use(express.json());
```

Once we have its reference, we just have to tell the app what parsing method we will use. There are options, such as text, JSON and url encoding.

- For now we will settle only on the last one with the following settings

```
app.use(express.urlencoded({ extended:true }));
```

Some utilities

This way we will be able to *parse* and *access* the variables received in the request

The config line is written right after the creation of the *app* variable.

Once it is configured, we will be able to access the variables inside our request through the body property.

req.body.{variable_name}

The *variable_name* comes from the name we set into the fields we create in the html side.

- E.g. req.body.num1

The values obtained that way are read as text, thus if I intend to receive some information in number format, I have to cast it with the **Number** function

- E.g. Number(req.body.num1)

Serving html

The goal of our app is to serve a fully developed website, but if we would have to write all the html into strings to later send it at response, the express files would be humongous.

Thankfully, express allows you to serve html files that are stored in your backend to the front end with the **sendFile** method on the *response* object.

This method will get a file from the server side and send it directly in the response

The problem that might arise with this method is that in order to serve the proper file, you have to know the **path** to it. And it might not be possible to have that path in every situation.

To solve that problem, you can use the **__dirname** variable. This variable is an environment variable that gives you the path to the local root of your app and allows you to concatenate the rest of the path to serve the page.

```
res.sendFile(__direname + ”/path/to/file.html”);
```

Serving other files

The **__dirname** variable serves well if you will only have some files to load. But in general this should not be the default way of serving static file.

To serve files we would regularly create a folder called public, where inside we would place our css, images, sounds, etc. folders

Then we just have to let express know that these will be served from there via the following configuration:

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

Inside this folder we should follow the same structure we used to create. This will be crucial when serving files to the html such as the css file or any specialized script or resource.

app.post

Previously we learned about the difference between GET and POST methods when learning about forms in html.

Express uses these same two method to receive data.

The syntax for the post method is very similar to that of the get method.

app.post("<*route_path*>", (req, res) => { <callback_function_body> });

In a *post* method, you would be expecting to receive information from the user, so we will use the ***body-parser*** module that we introduce earlier to retrieve the required data.

Just as we did in the *get*, this method will return the information in the *res* object.

Binding data

So far we have not really looked at how can we send the information from the front end to the back end. The most we got to was to introduce the HTML forms.

And those are precisely what we will use. To configure a form to connect to our backend we have to set two attributes.

- action : should point to the route that will capture this request. E.g “/”, “/contact”, etc.
- method: should select what type of method will be receiving the request, either a POST or GET method.

The method part is very important, since it is common that novice programmers do not understand how it works and some requests are just being rejected because either the get or post method do not respond to the expected requests.

Once the form is configured as such, the other thing we have to be aware is the name attribute of all the input fields. These names are going to be the ones we will use to access their posted value in the backend, using the **body-parser** module

Router

We have discussed how we can create multiple endpoints and how we can direct these paths to the calling method (GET, POST, etc)

This may be good enough, but for maintenance purposes this could become a nightmare. Therefore Express provides another mechanism to deal with that and chain multiple handlers with a single path.

```
app.route("<route_path>")
    .get(   (req, res) => { <callback_function_body> } ),
    .post( (req, res) => { <callback_function_body> } );
```

Binding data

This code will read *var1* and *var2* from the form and return it as a concatenated string.

```
10      <form action="/" method="POST">
11          <input type="text" name="var1">
12          <input type="text" name="var2">
13          <button type="submit" name="submit">Calculate</button>
14      </form>
```

```
11      app.post("/", (req, res) => {
12          var var1 = req.body.var1;
13          var var2 = req.body.var2;
14          res.send(var1 + " - " + var2 );
15      });
```

It is important to point out that when the values are read from the request, they are read in string format, so to operate them, we will have to parse them.

Embedded JavaScript Templates

Now that we are able to receive information from the front end, we can close the loop using template technology.

This is not any new technology, but the JS environment provides multiple options. The most simple one is called Embedded JavaScript Templates (EJS).

EJS templates will give us the capability of mixing html with backend values by binding the information to be rendered in the front end.

To use it, we have to do 2 steps, first configure, then use. This package is already part of the core.

To configure we have to let our express server we will be using it. To do so we will need to add the following lines after we create the server, but before the routers.

```
app.engine('html', require('ejs').renderFile);

app.set('view engine', 'html');
```

Embedded JavaScript Templates

The next step is to create our templates in html format.

Templates are just regular html with special tags to point out where will we bind our backend data to be rendered in the page.

The tags are just the name of the given variable from the backend, surrounded by **<%= … %>**

```
<body>
   <h1>Success!!</h1>
   <h3>Welcome <%= name %></h3>
</body>
```

Embedded JavaScript Templates

Now that our template is ready we have to bind the information from the backend.

To do so, we will use a new method from the response object, it is called render.

This method will receive as parameters a template and a dictionary that will hold a series of key, value pairs where the key should be the label tag in the template and the value is the variable in our server that holds the information we want to display

```
var name = req.body.name;

res.render(__dirname+"/html/success.html", { name: name });
```

Once this method is called, our html returned will have the values substituted

Embedded JavaScript Templates

The previous method allows you to have any structure in your project. But this can be simplified if you follow the EJS suggested structure.

In order to find the templates, EJS suggests that you create a views folder, where inside you will place a series of .ejs files, where the name of such file will be the name of the template.

e.g. views/list.ejs => res.render(“list", { name: name });

The content of the .ejs file would be as we described earlier.

Embedded JavaScript Templates

It is important to point out that the variables that we use in the templates' display **HAVE** to be present when we pass them to render. If we don't have the value available we should include it as empty. We can have missing variables if they are skipped by a previous condition in the rendering workflow.

EJS allows you to embed some logic in your templates with control structures that can make them very dynamic.

The control structures will use a program-like syntax, but will be surrounded by the **<% %>** tags

We will just talk about the two basic control structures (if & loops), but check the documentation in order to learn the full potential of these templates (link in the references section)

Embedded JavaScript Templates

IF STATEMENT

As in any language the **if… else if … else** is available. This structure will evaluate Boolean conditions and will execute when the conditions are met.

```
<% if (names.length == 1) { %>
   Welcome <%= names[0] %>
<% } else if (names.length == 2) { %>
   Welcome you two <%= names[0] %>, <%= names[1] %>
<% } else { %>
   …
<% } %>
```

Embedded JavaScript Templates

FOR STATEMENT

The **for** statement follows a C-style structure.

```
<% for(var i=0; i<names.length; i++) {%>
        <%= names[i]  %>,
<% } %>
```

The looped through variable should be an iterable, if this is not true, then this will crash.

Embedded JavaScript Templates

We should notice in the syntax that the sections are delimited by braces **{ }**

All these have to be also surrounded by the EJS's tag delimiters **<% %>**

EJS also support other tags for specific functions as detailed below.

- **<%** 'Scriptlet' tag, for control-flow, no output
- **<%_** 'Whitespace Slurping' Scriptlet tag, strips all whitespace before it
- **<%=** Outputs the value into the template (HTML escaped)
- **<%-** Outputs the unescaped value into the template
- **<%#** Comment tag, no execution, no output
- **<%%** Outputs a literal '<%'
- **%>** Plain ending tag
- **-%>** Trim-mode ('newline slurp') tag, trims following newline
- **_%>** 'Whitespace Slurping' ending tag, removes all whitespace after it

Advanced Node.js

Node modules

As in any programming language, it is smart to break our code into smaller pieces that we can later reuse, mix and match in order to perform new tasks.

In the context of Node.js we achieve this by creating new modules.

Modules are just pieces of code that are open to be called from external locations. To do so we have to declare what will be the entry point for the code.

This is how a module is created:

```
module.exports.getDate = getDate;

function getDate() {  return <some_date>;  }


module.exports.getDay = getDay;

function getDate() {  return <some_day_value>;  }
```

Node modules

The declared functions ***getDate*** and ***getDay*** are the actual functions we want to call from the outside. These can be as complex as needed.

We declare the access to the functions to the outside world with the ***module.exports*** declaration**.**

- This will create an object that later will be used by the caller object to access the methods

```
module.exports.getDate = getDate;

function getDate() {
        return <some_date>;
}

module.exports.getDay = getDay;

function getDay() {
        return <some_day_value>;
}
```

Node modules

The part after the **module.exports** IS the name that we will use to access the function.

The part after the **equals sign** refers to the name of the function we are calling.

IMPORTANT: When creating the reference we DO NOT use the parenthesis of the function, that would execute the function rather than creating a reference.

The name of the handler and the function itself DO NOT have to be the same.

```
module.exports.getDate = getDate;

function getDate() {
        return <some_date>;
}

module.exports.getDay = getDay;

function getDay(param1) {
        return <some_day_value>;
}
```

Node modules

IMPORTING CUSTOM MODULES

If we created our own modules and we want to use them we will need to import them and then use the reference to call the method handler.

To require them we will need a full reference to the containing file:

const date = require(__dirname + "/modules/date.js");

Now that we have the reference, we can call the methods:

const date = date.getDate();

const day = date.getDay(3);

Observe how now we do use the parenthesis in order to call the function.

Happy Coding!