

# Node.js for PHP developers

This article was originally published at

<https://gist.github.com/joepie91/87c5b93a5facb4f99d7b2a65f08363db>. It has not been finished yet, but still contains some useful pointers.

## Learning a second language

If PHP was your first language, and this is the first time you're looking to learn another language, you may be tempted to try and "make it work like it worked in PHP". While understandable, this is a **really bad idea**. Different languages have fundamentally different designs, with different best practices, different syntax, and so on. The result of this is that different languages are also better for different usecases.

By trying to make one language work like the other, you get the **worst of both worlds** - you lose the benefits that made language one good for your usecase, and add the design flaws of language two. You should always aim to learn a language *properly*, including how it is commonly or optimally used. Your code is going to look and feel considerably different, and that's okay!

Over time, you will gain a better understanding of how different language designs carry different tradeoffs, and you'll be able to get the *best* of both worlds. This will take time, however, and you should always start by learning and using each language *as it is* first, to gain a full understanding of it.

One thing I explicitly recommend against, is [CGI-Node](#) - you should **never, ever, ever use this**. It makes a lot of grandiose claims, but it actually just reimplements some of the worst and most insecure parts of PHP in Node.js. It is also completely unnecessary - the sections below will go into more detail.

## Execution model

The "execution model" of a language describes how your code is executed. In the case of a web-based application, it decides how your server goes from "a HTTP request is coming in", to "the application code is executed", to "a response has been sent".

PHP uses what we'll call the "CGI model" to run your code - for every HTTP request that comes in, the webserver (usually Apache or nginx) will look in your "document root" for a `.php` file with the same path and filename, and then execute that file. This means that for every new request, it effectively starts a new PHP process, with a "clean slate" as far as application state is concerned. Other than `$_SESSION` variables, all the variables in your PHP script are thrown away after a response is sent.

This "CGI model" is a somewhat unique execution model, and only a few technologies use it - PHP, ASP and ColdFusion are the most well-known. It's also a very fragile and limited model, that makes it easy to introduce security issues; for example, "uploading a shell" is something that's only possible because of the CGI model.

Node.js, however, uses a different model: the "long-running process" model. In this model, your code is not executed *by* a webserver - rather, your code *is* the webserver. Your application is only started once, and once it has started, it will be handling an essentially infinite amount of requests, potentially hundreds or thousands at the same time. Almost every other language uses this same model.

This also means that your application state *continues to exist* after a response has been sent, and this makes a lot of projects much easier to implement, because you don't need to constantly store every little thing in a database; instead, you only need to store things in your database that you actually intend to store for a long time.

Some of the advantages of the "long-running process" model (as compared to the "CGI model"):

- You can share information between requests *without* having to store it in an external database or the session data.
- There is a lot less overhead per request, and you can handle more concurrent requests on the same server.
- You can continue doing work *after* having sent a response to the client, and there is no time limit.
- You can easily implement something that needs a long-running connection, such as applications that are based on WebSockets.
- It's not possible for an attacker to "upload a shell".

The reason attackers cannot upload a shell, is that there is no direct mapping between a URL and a location on your filesystem. Your application is *explicitly* designed to only execute specific files that are a part of your application. When you try to access a `.js` file that somebody uploaded, it will just send the `.js` file; it won't be executed.

There aren't really any disadvantages - while you do have to have a Node.js process running at all times, it can be managed in the same way as any other webserver. You can also use another webserver in front of it; for example, if you want to host multiple domains on a single server.

# Hosting

Node.js applications will not run in most shared hosting environments, as they are designed to *only* run PHP. While there are some 'managed hosting' environments like Heroku that claim to work similarly, they are usually rather expensive and not really worth the money.

When deploying a Node.js project in production, you will most likely want to host it on a VPS or a dedicated server. These are full-blown Linux systems that you have full control over, so you can run any application or database that you want. The cheapest option here is to go with an "unmanaged provider".

Unmanaged providers are providers whose responsibility ends at the server and the network - they make sure that the system is up and running, and from that point on it's your responsibility to manage your applications. Because they do not provide support for your projects, they are a lot cheaper than "managed providers".

My usual recommendations for unmanaged providers are (in no particular order): [RamNode](#), [Afterburst](#), [SecureDragon](#), [Hostigation](#) and [RAM Host](#). Another popular choice is [DigitalOcean](#) - but while their service is stable and sufficient for most people, I personally don't find the performance/resources/price ratio to be good enough. I've also heard good things about [Linode](#), but I don't personally use them - they do, however, apparently provide limited support for your server management.

As explained in the previous section, your application *is* the webserver. However, there are some reasons you might still want to run a "generic" webserver in front of your application:

- Easier setup of TLS ("SSL").
- Multiple applications for different domains, on the same server ("virtual hosts").
- Slightly faster static file serving.

My recommendation for this is [Caddy](#). While nginx is a popular and often-recommended option, it's considerably harder to set up than Caddy, especially for TLS.

# Frameworks

(this section is a work in progress, these are just some notes left for myself)

- execution model
- Express
- small modules

# Templating

If you've already used a templater like Smarty in PHP, here's the short version: use either [Pug](#) or [Nunjucks](#), depending on your preference. Both auto-escape values by default, but I strongly recommend Pug - it understands the actual structure of your template, which gives you more flexibility.

If you've been using `include()` or `require()` in PHP along with inline `<?php echo($foobar); ?>` statements, here's the long version:

The "using-PHP-as-a-templater" approach is quite flawed - it makes it very easy to introduce security issues such as [XSS](#) by accidentally forgetting to escape something. I won't go into detail here, but suffice to say that this is a serious risk, *regardless* of how competent you are as a developer. Instead, you should be using a templater **that auto-escapes values by default, unless you explicitly tell it not to**. [Pug](#) and [Nunjucks](#) are two options in Node.js that do precisely that, and both will work with Express out of the box.

---

Revision #1

Created 11 December 2024 18:31:42 by joepie91

Updated 11 December 2024 18:44:19 by joepie91