5. 12 FACTOR APPS IN PLAIN ENGLISH

Popular platform-as-a-service provider [Heroku](https://www.heroku.com/) (now a subsidiary of [Salesforce](http://www.salesforce.com/)…fancy that…) maintains a manifesto of sorts called [The Twelve-Factor App](http://12factor.net/). It outlines a methodology for developers to follow when building modern web-based applications. Despite being partly self-serving (apps built like this will translate more naturally to running on Heroku), there’s a lot of meaty best-practices worth examining.

**Strive for These Best Practices**

I think these concepts are important for readers of ClearlyTech, even if you aren’t the developer! For those who desire to know why this stuff is important, or who want to have an intelligent conversation with their development team about these issues, I present to you **12-Factor Apps in Plain English**:

[**I. Codebase**](http://12factor.net/codebase)**— *One codebase tracked in revision control, many deploys***

Put all your code in a source control system. Heck, just put it up on [GitHub](http://www.github.com/) from the start.

All your application code lives in one repository. Once you get big, you may have a distributed system with multiple apps talking to each other (like a web application and a backend API), at which point you can treat them as separate apps with their own code repositories (still in source control, of course).

A codebase is run by developers on their local machines, and deployed to any number of other environments, like a set of testing machines, and the live production servers.

**Importance: Non-negotiable** *Everyone does this, and developers will laugh at you if you aren’t.*

[**II. Dependencies**](http://12factor.net/dependencies)**— *Explicitly declare and isolate dependencies***

All the environments your code runs in need to have some dependencies, like a database, or an image processing library, or a command-line tool. Never let your application assume those things will be in place on a given machine. Ensure it by baking those dependencies into your software system.

Most languages and frameworks provide a natural way to do this. You list all the versions of all the libraries you expect to have in place, and when the code is deployed, a command is run to download all the right versions and put them in place. No guesswork, everything as it needs to be.

This philosophy extends to your devs or devops team managing entire machine configurations using management tools like [Chef](http://www.getchef.com/chef/) and [Puppet](http://puppetlabs.com/)

**Importance: High** *Without this, your team will have a constant slow time-suck of confusion and frustration, multiplied by their size and number of applications. Spare yourself.*

[**III. Config**](http://12factor.net/config)**— *Store config in the environment***

Configuration is anything that may vary between different environments. Code is all the stuff that doesn’t.

The code that talks to your database will always be the same. But the location of that database (which machine it’s running on) will be different for a local developer machine than it will for your production servers. Likewise, in your testing environment, you might want to log debugging information about each web request, but in production that would be overkill.

Usernames and passwords for various servers and services also count as configuration, and should never be stored in the code. This is especially true because your code is in source control (see I. above) which means that anyone with access to the source will know all your service passwords, which is a bad security hole as your team grows.

All configuration data should be stored in a separate place from the code, and read in by the code at runtime. Usually this means when you deploy code to an environment, you copy the correct configuration files into the codebase at that time.

**Importance: Medium** *Lots of companies get away without this, but you’re sloppy if you do.*

[**IV. Backing Services**](http://12factor.net/backing-services)**— *Treat backing services as attached resources***

Your code will talk to many services, like a database, a cache, an email service, a queueing system, etc. These should all be referenced by a simple endpoint (URL) and maybe a username and password. They might be running on the same machine, or they might be on a different host, in a different datacenter, or managed by a cloud SaaS company. The point is, your code shouldn’t know the difference.

This allows great flexibility, so someone from your team could replace a local instance of Redis with one served by Amazon through Elasticache, and your code wouldn’t have to change.

This is another case where defining your dependencies cleanly keeps your system flexible and each part is abstracted from the complexities of the others…a core tenet of good architecture.

**Importance: High** *Given the current bindings to services, there’s little reason not to adhere to this best-practice.*

[**V. Build, release, run**](http://12factor.net/build-release-run)**— *Strictly separate build and run stages***

The process of turning the code into a bundle of scripts, assets and binaries that run the code is the *build*. The *release* sends that code to a server in a fresh package together with the nicely-separate config files for that environment (See III. above). Then the code is *run* so the application is available on those servers.

The idea here is that the build stage does a lot of heavy lifting, and developers manage it. The run stage should be simple and bullet-proof so that your team can sleep soundly through the night, knowing that the application is running well, and that if a machine gets restarted (say, a power failure happens) that the app will start up again on launch without the need for human intervention.

**Importance: Conceptual** *From a practical perspective, the tools and framework you use will define best-practices for building, deploying, and running your app. Some do a better job than others of enforcing strict separation, but you should be okay if you follow your framework’s suggested mechanisms.*

[**VI. Processes**](http://12factor.net/processes)**— *Execute the app as one or more stateless processes***

It’s likely you will have your application running on many servers, because that makes it more fault tolerant, and because you can support more traffic. As a rule, you want each of those instances of running code to be *stateless*. In other words, the state of your system is completely defined by your databases and shared storage, and not by each individual running application instance.

Let’s say you have a signup workflow, where a user has to enter 3 screens of information to create their profile. One (wrong) model would be to store each intermediate state in the running code, and direct the user back to the same server until the signup process is complete. The right approach is to store intermediate data in a database or persistent key-value store, so even if the web server goes down in the middle of the user’s signup, another web server can handle the traffic, and the system is none-the-wiser.

**Importance: High** *Not only is a stateless app more robust, but it’s easier to manage, generally incurs fewer bugs, and scales better.*

[**VII. Port binding**](http://12factor.net/port-binding)**— *Export services via port binding***

We’re getting a bit technical now, but stick with me. This factor is an extension of factor IV. above. The idea is that, just like all the backing services you are consuming, *your* application also interfaces to the world using a simple URL.

Usually you get this for free because your application is already presenting itself through a web-server. But let’s say you have an API that’s used by both your customers in the outside world (untrusted) and your internal website (trusted). You might create a separate URL to your API that your website can use which doesn’t go through the same security (firewall and authentication), so it’s a bit faster for you than for untrusted clients.

**Importance: Medium** *Most runtime frameworks will give you this for free. If not, don’t sweat it. It’s a clean way to work, but it’s generally not hard to change later.*

[**VIII. Concurrency**](http://12factor.net/concurrency)**— *Scale out via the process model***

When running your code, the idea is that lots of little processes are handling specific needs. So you might have dozens of handlers at the ready to process web requests, and another dozen to handle API calls for your enterprise users. And still another half-dozen processing background welcome-emails going to new users, or sending tweets for your users sharing things on your social media service.

By keeping all these small parts working independently, and running them as separate processes (in a low-level technical sense), your application will scale better. In particular, you’ll be able to do more stuff *concurrently*, by smoothly adding additional servers, or additional CPU/RAM and taking full advantage of it through the use of more of these small, independent processes.

**Importance: Low** *Don’t worry about this factor until you get pretty deep into scaling considerations. Trust your chief architect or CTO to raise the red flag if this is going to become an issue for you.*

[**IX. Disposability**](http://12factor.net/disposability)**— *Maximize robustness with fast startup and graceful shutdown***

When you deploy new code, you want that new version to launch right away and start to handle traffic. If an application has to do 20 seconds of work (say, loading giant mapping files into RAM) before it’s ready to handle real traffic, you’ve made it harder to rapidly release code, and you’ve introduced more churn on the system to stop/start independent processes.

With the proliferation of so many 3rd party libraries in today’s software systems, sub–1-second startup times are less and less common. But beyond loading code, your application should have everything it needs waiting in high-speed databases or caches, so it can start up snappily and be ready to serve requests.

Further, your application should be robust against crashing. Meaning, if it does crash, it should always be able to start back up cleanly. You should never do any mandatory “cleanup” tasks when the app shuts down that might cause problems if they failed to run in a crash scenario.

**Importance: Medium** *Depending on how often you are releasing new code (hopefully many times per day, if you can), and how much you have to scale your app traffic up and down on demand, you probably won’t have to worry about your startup/shutdown speed, but be sure to understand the implications for your app.*

[**X. Dev/prod parity**](http://12factor.net/dev-prod-parity)**— *Keep development, staging, and production as similar as possible***

It has become *in vogue* in recent years to have a much more rapid cycle between developing a change to your app and deploying that change into production. For many companies, this happens in a matter of hours. In order to facilitate that shorter cycle, and the risk that something breaks when entering production, it’s desirable to keep a developer’s local environment as similar as possible to production.

This means using the same backing services, the same configuration management techniques, the same versions of software libraries, and so on.

This is often accomplished by letting developers use a tool like [Vagrant](http://www.vagrantup.com/) to manage their own personal virtual server that’s configured just like production servers.

**Importance: Medium** *Developers will feel like taking shortcuts if their local environment is working “well enough”. Talk them out of it and take a hard-line stance instead, it’ll pay off long-term.*

[**XI. Logs**](http://12factor.net/logs)**— *Treat logs as event streams***

Log files keep track of a variety of things, from the mundane (your app has started successfully) to the critical (users are receiving thousands of errors).

In an ideal situation, those logs are viewed by developers in their local consoles, and in production they are automatically captured as a stream of events and pushed into a real-time consolidated system for long-term archival and data-mining like Hadoop.

At the very least, you should be capturing errors and sending them to an error reporting service like [New Relic](http://newrelic.com/) or [AirBrake](https://airbrake.io/). You can take a more general approach and send your logs to a service like [PaperTrail](https://papertrailapp.com/) or [Splunk Storm](https://www.splunkstorm.com/).

**Importance: Low** *If you are relying on logs as a primary forensic tool, you are probably already missing out on better solutions. Be sure to consolidate your logs for convenience, but beyond that, don’t worry about being a purist here.*

[**XII. Admin processes**](http://12factor.net/admin-processes)**— *Run admin/management tasks as one-off processes***

You’ll want to do lots of one-off administrative tasks once you have a live app. For example, doing data cleanup on bad data you discover; running analytics for a presentation you are putting together, or turning on and off features for A/B testing.

Usually a developer will run these tasks, and when they do, they should be doing it from a machine in the production environment that’s running the latest version of the production code. In other words, run one-off admin tasks from an identical environment as production. Don’t run updates directly against a database, don’t run them from a local terminal window.

**Importance: High** *Having console access to a production system is a critical administrative and debugging tool, and every major language/framework provides it. No excuses for sloppiness here.*

**Summary**

Some of these items may seem esoteric, as they are rooted in some fundamental systems design debates. But at the heart of a happily running system is an architecture that is robust, reliable, and surprises us as little as possible. These 12 factors are being adopted by most major software platforms and frameworks, and to cut corners against their grain is a bad idea. Discuss these issues with your development team, see if there are some quick wins to improve the quality of your application design.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*