12-factor-app

Jonas Thorhauge Grønbek

December 15, 2020

Abstract

Modern applications often utilize frameworks like the 12-factor app because of generalized community consensus. Strict general frameworks like the 12-factor app surely provides structure, but unspecific architectures can lead unnecessary or more critically - inappropriate code-bases. I will consider the twelve-factor app's twelve rules, and will reorder the rules and elaborate on why some of them might be even harmful.

1 The twelve factors

The twelve factors have been created in order to improve or solve multiple complications specifically related to software development. Adam Wiggins has created the twelve-factor framework in order to improve on traditional challenges of the implementation and denotes that the usage of his framework should conform to the following;
[1, Use declarative formats for setup automation, to minimize time and cost for new developers joining the project
Have a clean contract with the underlying operating system, offering maximum portability between execution environments
Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration
Minimize divergence between development and production, enabling continuous deployment for maximum agility
And can scale up without significant changes to tooling, architecture, or development practices.]
The following practices are highlighted and structured throughout 12 rules, which are to be conformed to in order to utilize the twelve-factor framework

[2, I. Codebase
One codebase tracked in revision control, many deploys
II. Dependencies
Explicitly declare and isolate dependencies
III. Config
Store config in the environment
IV. Backing services
Treat backing services as attached resources
V. Build, release, run

Strictly separate build and run stages
VI. Processes
Execute the app as one or more stateless processes
VII. Port binding
Export services via port binding
VIII. Concurrency
Scale out via the process model
IX. Disposability
Maximize robustness with fast startup and graceful shutdown
X. Dev/prod parity
Keep development, staging, and production as similar as possible
XI. Logs
Treat logs as event streams
XII. Admin processes
Run admin/management tasks as one-off processes]

2 Relevancy of the twelve factors

I will address the twelve factors ordered relative to the section header.

2.1

Regarding the very first rule it is suggested that the whole code-base is to be stored at a central repository. One code-base is not regarded as a standard approach when it comes to micro-services, and companies like NGINX has touched specifically on the matter of the twelve-factor rules in regards to the one code-base principle.
[3, The Twelve-Factor App recommends one codebase per app.
In a micro-services architecture, the correct approach is actually one codebase per service. Additionally, we strongly recommend the use of Git as a repository, because of its rich feature set and enormous ecosystem]. [4]I believe this to be downright unmanageable for big tech companies considering they have possible thousands of services with multiple programming languages utilized

2.2

[1]The second principle regards isolating dependencies of the project. [3]is well regarded as a good practice in modern software-development and has been a genuine complication of older monolithic systems.

2.3

The third principle regards storing configuration in the local environments. I don't think this requires a dedicated explanation since it has been the standard for a long time.

2.4

The fourth principle regards backing services [1, A backing service is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as MySQL or CouchDB), messaging/queueing systems (such as RabbitMQ or Beanstalkd)]. These backing services should be attached or detached to deploys at will. By this I assume it would be done by either dependency injections at compile-time or be isolated in services that could deploy individually to change them at run-time. The ladder really appeals to me, since I can definitely see the benefits in large distributed systems where one might decide to change log-servers or databases.

2.5

The fifth principle regards building, running and releasing code. This does not need much explanation as development on production sites typically include staging builds, feature builds and production builds accompanied by CI/CD.

2.6

The sixth principle regards as one or more stateless processes. This concept is really welcomed, as it makes for a more reliable system across the different environments. [3, For microservices, the important point in the Processes factor is that your application needs to be stateless. This makes it easy to scale a service horizontally by simply adding more instances of that service. Store any stateful data, or data that needs to be shared between instances, in a backing service.]

2.7

The seventh principle regards port binding. This is a common way to structure modern applications used by resource management technologies like Docker, docker-compose or Kubernetes.

2.8

The eight principle regards concurrency. This is not referring to the usage of multi-threading but rather how to structure the different processes within your application. [1, Any computer program, once run, is represented by one or more processes. Web apps have taken a variety of process-execution forms. For example, PHP processes run as child processes of Apache, started on demand as needed by request volume. Java processes take the opposite approach, with the JVM providing one massive uberprocess that reserves a large block of system resources (CPU and memory) on startup, with concurrency managed internally via threads. In both cases, the running process(es) are only minimally visible to the developers of the app.]
This generally seems like a reasonable way of structuring your processes, and handling it via the hosts process manager is a welcomed decision. This is handled by default if utilizing containerization. [3, With containerized services, you further get the concurrency recommended in the Twelve-Factor App, for free.]

2.9

The ninth principle regards robustness of startup and shutdowns. [3]This generally seems to be a default for any application that utilizes docker as the previous principle.

2.10

the tenth principle regards having different environments be as similar as possible. [1, Developers sometimes find great appeal in using a lightweight backing service in their local environments, while a more serious and robust backing service will be used in production. For example, using SQLite locally and PostgreSQL in production; or local process memory for caching in development and Memcached in production.]. This is a principle I think most conforms to without thinking too much about it, and it suits the other principles perfectly well.

2.11

the eleventh principle denotes the usage of logs and how they should be treated. The twelve-factor states that it should be treated as a stream. This is with the purpose of enabling or improving the following:

- Finding specific events in the past.
- Large-scale graphing of trends (such as requests per minute).
- Active alerting according to user-defined heuristics (such as an alert when the quantity of errors per minute exceeds a certain threshold).

This seems perfectly reasonable and I would further add that they should even be considered in the PaaS strategy should one be utilized.

2.12

The twelfth and last principle by the twelve-factor app indicates that admin/management tasks should be run as one-off processes. This is also something that fits the modern micro-service architecture perfectly since containers make this very easy, as you can spin up a container just to run a task and then shut it down.

3 Conclusion

I would only consider a single principle to be - strictly speaking harmful for specific applications. That is the very first principle that denotes that applications should store the whole code-base in a single repository. Based on the information I have provided throughout my article I would order the principles as follows from the perspective of a larger company that utilizes modern paradigms like micro-service-oriented architectures.
5

1. principle 9: Maximize robustness with fast startup and graceful shutdown.
2. principle 2: Explicitly declare and isolate dependencies.
3. principle 12: Run admin/management tasks as one-off processes
4. principle 5: Strictly separate build and run stages

5. principle 10: Keep development, staging, and production as similar as possible.

6. principle 6: Execute the app as one or more stateless processes.

7. principle 4: Treat backing services as attached resources.

8. principle 8: Scale out via the process model.

9. principle 3: Store config in the environment

10. principle 7: Export services via port binding.

11. principle 12: Run admin/management tasks as one-off processes

12. principle 1: One code-base tracked in revision control, many deploys

References

[1] Adam Wiggins. Twelve-factor official documentation. https://web.archive.org/web/20170613060854/https://12factor.net/12factor.epub.

[2] Adam Wiggins. The twelve-factor app. https://12factor.net/, 2017.

[3] NGINX. Mra, part 5: Adapting the twelve-factor app for microservices. https://www.nginx.com/blog/microservices-reference-architecture-nginx-twelve-factor-app/, 2016.

[4] Wikipedia. Programming languages used in most popular websites. https://en.wikipedia.org/wiki/Programming$_languages_used_in_most_popular_websites$, 2020.