All six blogs, plus a blog about web frontends for microservices applications, have been collected into a free ebook.

Also check out these other NGINX resources about microservices:

- *A very useful and popular series by Chris Richardson about microservices application design*
- *The Chris Richardson articles collected into a free ebook, with additional tips on implementing microservices with NGINX and NGINX Plus*
- *Other microservices blog posts*
- *Microservices webinars*

The Twelve-Factor App

Software is increasingly delivered over the Internet as a service. Originally called Software as a Service (SaaS), similar software – with a much stronger emphasis on mobile interaction – is now usually referred to as web apps.

The Twelve-Factor App is a praiseworthy effort by Heroku, a platform as a service (PaaS) provider, to establish general principles for creating useful web apps. However, the original principles are somewhat specific to Heroku's PaaS platform. They aren't an exact fit for a microservices architecture.

In implementing the NGINX MRA, we've extended the Twelve-Factor App with our own additions and microservices-specific modifications. We've found the amended version extremely useful.

2 – Dependencies

Explicitly declare and isolate dependencies

As suggested in The Twelve-Factor App, regardless of what platform your application is running on, use the dependency manager included with your language or framework. How you install operating system or platform dependencies depends on the platform:

- In noncontainerized environments, use a configuration management tool (Chef, Puppet, Ansible) to install system dependencies.
- In a containerized environment, do this in the Dockerfile.

Note: We recommend that you choose a dependency management mechanism in the context of your comprehensive Infrastructure-as-Code strategy, not as an isolated decision. See Martin Fowler's writings on Infrastructure-as-Code and download the O'Reilly report *Infrastructure as Code* by Kief Morris.

3 – Config

Store configuration in the environment

Anything that varies between deployments can be considered configuration. The Twelve-Factor App guidelines recommend storing all configuration in the environment, rather than committing it to the repository. We recommend the following specific practices:

- Use non-version controlled **.env** files for local development. Docker supports the loading of these files at runtime.
- Keep all **.env** files in a secure storage system, such as Vault, to keep the files available to the development teams, but not commited to Git.
- Use an environment variable for anything that can change at runtime, and for any secrets that should not be committed to the shared repository.

Part of F5

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.

7 – Data Isolation

Each service manages its own data

As a modification to make the Port binding factor more useful for microservices, we recommend that you allow access to thepersistent data owned by a service only via the service's API. This prevents implicit service contracts between microservices and ensures that microservices can't become tightly coupled. Data isolation also allows the developer to choose, for each service, the type of data store that best suits its needs.

8 – Concurrency

Scale out via the process model

The Unix process model is largely a predecessor to a true microservices architecture, insofar as it allows specialization and resource sharing for different tasks within a monolithic application. In a microservices architecture, you can horizontally scale each service independently, to the extent supported by the underlying infrastructure. With containerized services, you further get the concurrency recommended in the Twelve-Factor App, for free.

9 – Disposability

Maximize robustness with fast startup and graceful shutdown

Instances of a service need to be disposable so they can be started, stopped, and redeployed quickly, and with no loss of data. Services deployed in Docker containers satisfy this requirement automatically, as it's an inherent feature of containers that they can be stopped and started instantly. Storing state or session data in queues or other backing services ensures that a request is handled seamlessly in the event of a container crash. We are also proponents of using a backing store to support crash-only design.