Cloud Native Software Development

Software development doesn't exist in a vacuum. The software will need to run somewhere. While not every software needs to, or indeed will, scale to Google levels, a little forethought can go a long way towards surviving the first contact with the customer hordes.

The enemy of scalability: state

Any application that does not store or access data is easy to scale. Simply run more copies of it and put a load balancer in front of it.

State can rear its ugly head in obvious, and in more subtle ways. The obvious point state is, for example, your database. When you have data in your database that's a form of state. You can run as many application servers as you want, if the database can't handle the load you're done. As we discussed in the third lecture scaling databases can be quite tricky since the consistency model may not permit scaling.

A second point of contention is the ability to store data. Integrating an object storage instead of simply writing data to the disk will, again, help with scaling.

A more subtle point of contention may be the storage of sessions. Sessions are usually built by sending the user a session ID via a cookie and then storing a blob of data on the disk or in the database. The problem is not only the storage, but also that moving from a local filesystem to a database may create a race condition that may present a security issue with your application.

A similar issue arises when near-realtime data exchange is desired between a large amount of users (e.g. a chat application). While a single server can scale to tens of thousands of users, that server is neither redundant, nor will it serve an endless number of users. PubSub systems can help with that.

The Twelve Factor App

Once we survive the first night after the launch of our application it often becomes time to deal with long-term architectural problems. The 12 factor app is a concept that collects the current best practices of writing applications for the cloud. Keep in mind that these are just guidelines and you should never make a religious tyrade out of following these guidelines. Instead, apply common sense and be pragmatic about what to do.

1. Codebase

This first guideline is pretty simple: keep your application in a version control system. While this should be the default in 2020 it unfortunately still bears saying.

2. Dependencies

This recommendation deals with dependencies. Almost every programming language ecosystem nowadays has a dependency manager to handle downloading dependencies. These dependencies should be declared explicitly in the configuration file for the dependency manager (e.g. `package.json`, `composer.json`, `pom.xml`, etc.)

Furthermore, these dependency managers often create a lock file (`npm-package-lock.json` , `composer.lock` , etc.). They record the exact versions, and sometimes hashes of the third party library. This ensures that the same exact version is installed in the development and production builds.

3. Configuration

While in older times configuration files had many formats, modern, container-based applications distinctly move towards environment variables. Environment variables are a cross-platform way to provide variables to an application and give a DevOps engineer a flexible way to configure a containerized application.

4. Backing services

External databases, caches, etc. should be treated as such. This means that the connection options (server name, url, username, password, etc) should be configurable via environment variable and should also be replacable by a system administrator by simply reconfiguring the container.

Testing the testability is especially important with services that are not as well standardized as a MySQL database. For example, many S3 implementations only support Amazon's version and cannot be configured to use alternative providers. An application following this recommendation should be tested against at least one other S3 provider.

5. The build process

The build recommendation says that the build process should be separated from the runtime configuration and the release process. In practice this means that you don't bake your configuration into your container. It also means that each release should have a version number or date, allowing you to roll back to a specific version.

6. Stateless processes

The processes recommendation says that your application should run as a single program without any shared data between different copies of your application. Any state should be outsourced to an external database or system as mentioned before.

7. Network configuration

The port binding recommendation says that an application is standalone and does not require an external webserver to run. (Typically PHP applications require a bit of legwork to satisfy this requirement.)

8. Process handling

The concurrency recommendation defines a number of rules for process handling. It requires that it should be possible to run multiple copies of the application, optionally across several machines. It also requires that the application should not daemonize.

While this may seem as trivial with a green field project, it is definitely a challenge for legacy application.

9. Disposability

This recommendation states that an application should have a fast startup and should also shut down quickly when receiving the TERM signal. Applications should also be able to recover from unexpected crashes.

10. Development/production gap

The dev/prod parity recommendation postulates that developers of apps should strive to have a continuous method of deployment that allows for rapid rollout of updates. This lowers the work a developer needs to go through to deploy and therefore promotes smaller change sets.

Smaller change sets make tracking down issues with deployments easier, but can be impractical when working with larger, slower moving clients. (e.g. banks, telcos, etc.)

11. Logs

The logs recommendation should simply write logs to the standard output in a standardized form (e.g. JSON). Everything else, like routing or storing logs, is not the applications concern.

In practice each application contains at least some basic logic to filter logs to make it easier to configure the logging level.

12. Tooling

The admin process recommendation concerns itself with the command line tools needed to operate the application. These are things like running database migrations, etc. The recommendation says that these tools should be runnable directly from the applications directory without much extra configuration or installation.

Metrics collection

The above 12 factors are, by necessity, limited in scope. The author(s) of those 12 factors have taken many things into account, yet left out others.

One of these is metrics collection. As you will learn in the Prometheus exercise, Prometheus has established itself as a defacto standard for monitoring cloud-native applications.

It is not uncommon to see applications include a small webserver that exposes internal metrics in the Prometheus data format. This makes monitoring exceedingly easy compared to having to configure a separate monitoring service.

Health checks

When dealing with containerized environments it is very important that applications report accurately when they are ready to serve traffic, and when they are having problems.

This can be achieved by implementing the `HEALTHCHECK` directive in the `Dockerfile`, or by implementing Liveness, Readiness, and Startup Probles in Kubernetes.