



DevOps and the 12-Factor App

What is DevOps?

- DevOps originates as a cultural and professional movement that emphasizes collaboration and communication between software developers and IT professionals.
- The goal of DevOps is to automate and integrate the processes between these groups.
- Why do you want this? Faster delivery, improved deployment frequency, lower failure rates of new releases, etc.
- The core principles of DevOps are Continuous Integration (CI), Continuous Delivery (CD), and Infrastructure as Code (IaC).
- These principles facilitate a more agile and responsive development cycle, focused on rapid iterations and feedback.

Introduction to Twelve-Factor App

- The Twelve-Factor App methodology is a collection of best practices for building web applications that can be developed and deployed quickly, frequently, and reliably, which aligns with the goals of DevOps.
- What's nice about the TFA is that it covers a lot of best practices for DevOps, from managing your codebase to handling logs.
- Cloud technologies are often setup to already follow the Twelve-Factor app to some degree.
- Website: <https://12factor.net>

Deeper Dive into Each Factor

1. One codebase tracked in revision control, many deploys

- The first factor states that you should store the code in a version control system like Git.
- It's possible that your app consists of multiple services: your website (or frontend) server, a backend server, a database server, services for specific things like handling payments, performing analytics, etc.
- In the case of Leadspotr: frontend + backend.
- You can use separate repositories for each service. The advantage is that this allows you control over who has access to what part of your codebase.
- You can also combine the services in a single repository. The advantage is that everything is in a single place and it's easier to maintain consistency.
- If your apps and services need to share code (like certain data structures or interfaces), extract these into libraries that you can then include using a dependency manager. If you're building a service with JavaScript or Typescript and Node.js, you can use npm. In Python, you can use Pip, or more advanced tools for managing dependencies like Poetry.
- There is one codebase, but there can be multiple deploys. A deploy is a running instance of your app. The production version of your app is one of the deploys, but there can be other versions running as well: a version locally on your machine, a version used for testing, a staging version, and so on.
- Leadspotr has two deploys: one for running the app locally, and another for running the production app.

2. Explicitly declare all dependencies

- In most programming languages, you have a mechanism for handling the dependencies that your application needs. If you're doing Typescript or JavaScript development, then you'll probably use NPM or Yarn. Rust has the Cargo package manager. In Python, that's Pip.
- A 12-factor app doesn't rely on system-wide installed packages, but it declares all its dependencies using some kind of manifest file. In Typescript and JavaScript, this is the package.json file that contains all the dependencies. If you're using Pip in Python, this is the requirements text file. In Poetry, it's pyproject.toml. Because there's only a single codebase that's deployed one or more times to different environments, the specification of these dependencies is also the same for both production and development environments. Though of course, these can be attached to different branches in your Git repository.
- A second thing that's important for dealing with dependencies is that there's also a way to isolate dependencies so to ensure that you're not accidentally using dependencies from the surrounding system. In Python, you can use something like virtualenv to isolate your dependencies. If you want to take this a step further, there

are overarching dependency management and isolation systems like Poetry. I'm using Poetry for most of the examples in this course.

- But even then, there's another layer of isolation that you can benefit from, and that's the system-wide specification of dependencies and providing an isolated execution environment. The tool for this that everyone uses is obviously Docker. This is a light-weight virtualization layer that allows you to specify exactly which operating system and version you'll be using and installs any extra dependencies that you need as well. The Dockerfile is the manifest specifying the execution context and can install dependencies that you need. Docker containers are the virtual environments that provide the isolation. I'll talk more about Docker in the next lesson.

3. Store config in the environment

- Depending on the deploy of your app, you're going to have many configuration settings. These settings will be different for the different deploys. For example, your production web application will be on a different domain than your staging or development version of the app. And there are other things that might be different:
 - Connecting to the development database instead of the production database
 - Credentials for storage locations like Amazon S3 or Google Cloud storage
 - Credentials of other services like email clients or other third-party integrations
- You may even have globally enabled or disabled areas of your application depending on the deploy. For example, an admin area might not be visible by default in production, you could have features that you're already building but they're hidden behind a global feature flag, a global maintenance mode that is on or off, logging settings, and so on.
- These kinds of settings shouldn't be part of your codebase. If you're for example storing these things as constants in the code, this is a violation of the 12-factor app. A good test to verify that you're setting up configuration correctly is whether your code could be released as open source without giving public access to any credentials. Another test is to verify that if you give access to your repository to let's say an intern, that you can control what the intern has access to.
- The most common place to store these settings is by using environment variables. They're language and OS agnostic. Most cloud providers offer the possibility to define environment variables as part of your virtual server. For example, if you launch a Google Cloud function, you can add environment variables which are then available to the app at startup.
- You can also use a tool like dotenv to group your variables into a single dot env file. This works well for local development. Whenever a new developer wants to start working on the code, you can provide that developer a dot env file containing all the necessary credentials. But you must make sure that you don't accidentally commit the .env file to your repository.
- For some configuration settings it might make sense to make them part of the code. For example, the names of the routes in a web app, file size limits for uploads, default prefixes for database ids, etc. These things are probably not going to be different for different deploys, so you can keep them in the code itself. As soon as

they change depending on the deploy though, move them out into environment variables.

- Leadspotr: uses a .env file, which is also used in a GitHub Action

4. Treat backing services as attached resources

- Backing services are any service the app consumes over the network as part of its normal operation, such as datastores (like MySQL or MongoDB), messaging/queueing systems (like RabbitMQ or Kafka), and caching systems (like Memcached or Redis).
- From your app's perspective, both local and third-party services are treated as attached resources.
- Traditionally, applications were often tightly coupled with their backing services, leading to issues with portability and flexibility.
- Having hardcoded configurations or using local resources means that your app isn't easily scalable or replaceable.
- The Twelve-Factor methodology's treats backing services as attached resources. The "attachment" is a way of thinking about the relationship between the app and its backing services, which promotes loose coupling.
- Benefits:
 - Easy swapping of resources, such as switching databases without any code changes.
 - Greater flexibility in development, testing, and production environments as services can be attached or detached as needed.
- Implementation:
 - Storing resource handles in the environment config, making them easy to change. For example, the database connection string is an environment config value. As a result, you can easily swap out the database with another one.
 - Using consistent resource handles across all environments to avoid environment-specific configurations.

5. Separate build and run stages

- Deployment of your code should happen in three independent stages: build, release and run. In the build stage you convert your code into an executable bundle. It contains any dependencies and other assets that the code needs. If you use Docker, this would be the Docker image.
- Part of the build stage can be a test stage in which you run all sorts of tests to make sure the new code doesn't break anything. If it does, this should cancel the rest of the steps.
- You then have a release stage that combines any configuration settings with the build output. If you're using something like Kubernetes this is the definition of a deployment that links a Docker image with other settings like environment variables, information about the number of replicas and so on.

- Finally, there's the run stage that retrieves the Docker image and starts the service according to the release configuration (this part is normally hidden from you and done in the background).
- Every release should have a unique id: can be a timestamp, a commit id, or a version number. Once you create a release, it can't be mutated. Any change must create a new release.
- Separating build, release and run stages also means that you shouldn't change code in a service that's currently running. You change it in your code base, then build and release a new version.
- You can initiate a build and release after you've changed your code automatically. If you're using GitHub, then you can do this with GitHub Actions. I'll show you an example of that in the lesson covering CI/CD. If you're not using GitHub but for example BitBucket, they also have a similar feature called Pipelines.

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

- I've talked about stateful and stateless services in one of the previous lessons. This factor stresses that you need to distinguish the two. Examples of things that maintain state are your database, a cache, an object store, etc. Anything else shouldn't maintain state. For example if you develop an API service that acts as a layer on top of your database, it shouldn't have any state. You might be tempted to do things locally, like maintain a log file, or store user activity. Don't. Because if you ever decide to scale up and maintain multiple services in parallel, this can result in a big mess.
- Twelve-factor processes are stateless and share nothing. Any data that needs to persist must be stored in a stateful backing service, such as a database or an object store.
- Assume that at any moment, your stateless service could be restarted, and any local data will be lost.

7. Export services via port binding

- It's important to think about how applications expose their services to the outside world.
- "Port binding" is the process by which an application makes itself available to the network by binding to a port.
- Traditional model: applications were executed in an environment containing a web server, and the web server itself handled the task of exposing the application to the network.
- Result: lack of portability and difficulties in containerization.
- In the 12FA: applications are self-contained and export services by binding to a port.
- Containerization and Port Binding
 - Each container's internal ports can be mapped to arbitrary ports on the host, providing isolation and scalability.
 - Container orchestrators like Docker and Kubernetes use port binding to manage service discovery and routing.

8. Scale-out via process model

- Applications need to handle growth in users and traffic seamlessly.
- The traditional method is to scale up vertically by adding more resources to a single server.
- This is limited because you may not need those resources all the time, and you'll be limited by physical hardware.
- With 12FA, you will mostly scale horizontally, via the process model.
- The process model = executing multiple instances of the same application concurrently (which is easy if the processes are stateless).
- Benefits:
 - Improved resilience, as failure in one process does not impact others.
 - More efficient resource utilization, as processes can be distributed across multiple machines or cores.
 - Enhanced flexibility, as processes can be scaled independently based on workload requirements.
 - I.e. you can scale a web process for handling simple HTTP requests independently from worker processes that handle other tasks such as analytics.
- Example: Learntail (platform that generates quizzes using AI) will work with 2 different backend services: one for handling API requests (such as creating an account, logging in, etc). and one service for performing AI-related tasks: talking with an LLM, web scraping, document analysis, etc.

9. Fast startup and graceful shutdown

- Fast startup and graceful shutdown are crucial for scalability and reliability in modern cloud-based applications.
- Fast startup = the ability of an application to go from being launched to being ready to receive requests in the least amount of time possible.
- This contributes to better resource utilization, allowing for scaling out quickly and efficiently, especially in platforms that use containerization like Docker or Kubernetes.
- Best practices for fast startup:
 - Avoiding heavy lifting at startup time and deferring initialization of resources.
 - Building lean containers with only the necessary dependencies (multi-stage builds are helpful for this - see the Leadspotr code)
- Graceful shutdown = the ability of an application to handle termination signals in a way that it can finish current requests and clean up resources before shutting down.
- This avoids leaving requests unfulfilled and ensures data integrity.
- Best practices for graceful shutdown:
 - Capturing termination signals to execute shutdown procedures like releasing database connections (too many open connections can slow down your database)
 - Implementing health checks to prevent the routing of new requests to the shutting-down instance.

10. Keep development, staging, and production as similar as possible

- One thing that has really saved us a lot of time at my company is making sure that the various environments you deploy to are as similar as possible.
- And nowadays, we have lots of tools available to help us achieve that. Docker not only allows you to package up your application and its dependencies into a single isolated image, but you can also run that image on your local development machine.
- That way the locally running system will be much closer to when you deploy it. You should still have at least a staging environment though where you check that things also work as expected with a copy of the production data.

11. Treat logs as event streams

- Logs (= records of events that happen within an application) are important for monitoring and troubleshooting your applications.
- Traditionally logs were treated as files on disk. This has lots of issues: log rotation, log archival, and disk I/O limitations.
- In the 12FA, logs are seen as event streams.
- Logs are a stream of aggregated, time-ordered events collected from the output streams of all running processes and backing services.
- Benefits:
 - Logs become simple and lightweight, as applications do not concern themselves with routing or storage of their output stream.
 - Log indexing and searching are handled by specialized tools, which can provide better performance and flexibility.
- Implementation:
 - Applications can write their event streams to `stdout` for capture.
 - Mention the role of the execution environment in capturing these streams, adding metadata, and forwarding them to a log indexing and analysis system (for example Papertrail)
- Caveat with logging: be careful what you log! Accidentally exposed log files are common sources of data leaks (I'll talk more about this in a later lesson).

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

- Admin/management tasks are any tasks that are run occasionally or one-time at the behest of an admin or developer to perform operational work on an application.
- Examples could include database migrations, batch scripts, or one-time scripts to correct data in a production database.
- You might be tempted to run admin tasks directly through a live server / with cron jobs. Potential issues are lack of control, tracking, and varying runtime environments.
- With 12FA: admin/management tasks are separate, one-off processes.
- Characteristics of one-off admin processes:
 - Run against a release, using the same codebase and configuration as any process run against that release.

- Run in an environment as similar as possible to the regular long-running application.
 - Run in isolation, without interfering with the running application.
- Best practices:
 - Using version control for admin scripts to keep changes tracked (you can put this in a scripts folder in the GitHub repository)
 - Running tasks through a process manager, which handles logging and ensures the task is run in the app's current deployment environment.
 - Isolating one-off tasks to avoid any unintended side effects on the running application.

Summary

- DevOps and Twelve-Factor App are important methodologies for efficient and effective software development and deployment.
- DevOps is more than tools—it's a culture.
- Twelve-Factor App covers all aspects of an application from development to deployment and scaling.