3
How DevOps Affects Architecture

Software architecture is a vast subject, and in this book, we will focus on the aspects of architecture that have the largest effects on Continuous Delivery and DevOps and vice versa.

In this chapter, we will see:

- Aspects of software architecture and what it means to us while working with our DevOps glasses on
- Basic terminology and goals
- Anti-patterns, such as the monolith
- The fundamental principle of the separation of concerns
- Three-tier systems and microservices

We finally conclude with some practical issues regarding database migration.

It's quite a handful, so let's get started!

Introducing software architecture

We will discuss how DevOps affects the architecture of our applications rather than the architecture of software deployment systems, which we discuss elsewhere in the book.

Often while discussing software architecture, we think of the non-functional requirements of our software. By non-functional requirements, we mean different characteristics of the software rather than the requirements on particular behaviors.

A functional requirement could be that our system should be able to deal with credit card transactions. A non-functional requirement could be that the system should be able to manage several such credit cards transactions per second.

Here are two of the non-functional requirements that DevOps and Continuous Delivery place on software architecture:

- We need to be able to deploy small changes often
- We need to be able to have great confidence in the quality of our changes

The normal case should be that we are able to deploy small changes all the way from developers' machines to production in a small amount of time. Rolling back a change because of unexpected problems caused by it should be a rare occurrence.

So, if we take out the deployment systems from the equation for a while, how will the architecture of the software systems we deploy be affected?

The monolithic scenario

One way to understand the issues that a problematic architecture can cause for Continuous Delivery is to consider a counterexample for a while.

Let's suppose we have a large web application with many different functions. We also have a static website inside the application. The entire web application is deployed as a single Java EE application archive. So, when we need to fix a spelling mistake in the static website, we need to rebuild the entire web application archive and deploy it again.

While this might be seen as a silly example, and the enlightened reader would never do such a thing, I have seen this anti-pattern live in the real world. As DevOps engineers, this could be an actual situation that we might be asked to solve.

Let's break down the consequences of this tangling of concerns. What happens when we want to correct a spelling mistake? Let's take a look:

1. We know which spelling mistake we want to correct, but in which code base do we need to do it? Since we have a monolith, we need to make a branch in our code base's revision control system. This new branch corresponds to the code that we have running in production.
2. Make the branch and correct the spelling mistake.
3. Build a new artifact with the correction. Give it a new version.
4. Deploy the new artifact to production.

Okay, this doesn't seem altogether too bad at first glance. But consider the following too:

- We made a change in the monolith that our entire business critical system comprises. If something breaks while we are deploying the new version, we lose revenue by the minute. Are we really sure that the change affects nothing else?
- It turns out that we didn't really just limit the change to correcting a spelling mistake. We also changed a number of version strings when we produced the new artifact. But changing a version string should be safe too, right? Are we sure?

The point here is that we have already spent considerable mental energy in making sure that the change is really safe. The system is so complex that it becomes difficult to think about the effects of changes, even though they might be trivial.

Now, a change is usually more complex than a simple spelling correction. Thus, we need to exercise all aspects of the deployment chain, including manual verification, for all changes to a monolith.

We are now in a place that we would rather not be.

Architecture rules of thumb

There are a number of architecture rules that might help us understand how to deal with the previous undesirable situation.

The separation of concerns

The renowned Dutch computer scientist Edsger Dijkstra first mentioned his idea of how to organize thought efficiently in his paper from 1974, *On the role of scientific thought*.

He called this idea "the separation of concerns". To this date, it is arguably the single most important rule in software design. There are many other well-known rules, but many of them follow from the idea of the separation of concerns. The fundamental principle is simply that we should consider different aspects of a system separately.

The principle of cohesion

In computer science, cohesion refers to the degree to which the elements of a software module belong together.

Cohesion can be used as a measure of how strongly related the functions in a module are.

It is desirable to have strong cohesion in a module.

We can see that strong cohesion is another aspect of the principle of the separation of concerns.

Coupling

Coupling refers to the degree of dependency between two modules. We always want low coupling between modules.

Again, we can see coupling as another aspect of the principle of the separation of concerns.

Systems with high cohesion and low coupling would automatically have separation of concerns, and vice versa.

Back to the monolithic scenario

In the previous scenario with the spelling correction, it is clear that we failed with respect to the separation of concerns. We didn't have any modularization at all, at least from a deployment point of view. The system appears to have the undesirable features of low cohesion and high coupling.

If we had a set of separate deployment modules instead, our spelling correction would most likely have affected only a single module. It would have been more apparent that deploying the change was safe.

How this should be accomplished in practice varies, of course. In this particular example, the spelling corrections probably belong to a frontend web component. At the very least, this frontend component can be deployed separately from the backend components and have their own life cycle.

In the real world though, we might not be lucky enough to always be able to influence the different technologies used by the organization where we work. The frontend might, for instance, be implemented using a proprietary content management system with quirks of its own. Where you experience such circumstances, it would be wise to keep track of the cost such a system causes.

A practical example

Let's now take a look at the concrete example we will be working on for the remainder of the book. In our example, we work for an organization called Matangle. This organization is a software as a service (SaaS) provider that sells access to educational games for schoolchildren.

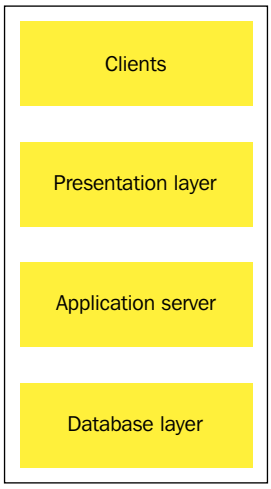
As with any such provider, we will, with all likelihood, have a database containing customer information. It is this database that we will start out with.

The organization's other systems will be fleshed out as we go along, but this initial system serves well for our purposes.

Three-tier systems

The Matangle customer database is a very typical three-tier, **CRUD** (**create, read, update,** and **delete**) type of system. This software architecture pattern has been in use for decades and continues to be popular. These types of systems are very common, and it is quite likely that you will encounter them, either as legacy systems or as new designs.

In this figure, we can see the separation of concerns idea in action:



The three tiers listed as follows show examples of how our organization has chosen to build its system.

The presentation tier

The presentation tier will be a web frontend implemented using the React web framework. It will be deployed as a set of JavaScript and static HTML files. The React framework is fairly recent. Your organization might not use React but perhaps some other framework such as Angular instead. In any case, from a deployment and build point of view, most JavaScript frameworks are similar.

The logic tier

The logic tier is a backend implemented using the Clojure language on the Java platform. The Java platform is very common in large organizations, while smaller organizations might prefer other platforms based on Ruby or Python. Our example, based on Clojure, contains a little bit of both worlds.

The data tier

In our case, the database is implemented with the PostgreSQL database system. PostgreSQL is a relational database management system. While arguably not as common as MySQL installations, larger enterprises might prefer Oracle databases. PostgreSQL is, in any case, a robust system, and our example organization has chosen PostgreSQL for this reason.

From a DevOps point of view, the three-tier pattern looks compelling, at least superficially. It should be possible to deploy changes to each of the three layers separately, which would make it simple to propagate small changes through the servers.

In practice, though, the data tier and logic tier are often tightly coupled. The same might also be true for the presentation tier and logic tier. To avoid this, care must be taken to keep the interfaces between the tiers lean. Using well-known patterns isn't necessarily a panacea. If we don't take care while designing our system, we can still wind up with an undesirable monolithic system.

Handling database migrations

Handling changes in a relational database requires special consideration.

A relational database stores both data and the structure of the data. Upgrading a database schema offers other challenges then the ones present when upgrading program binaries. Usually, when we upgrade an application binary, we stop the application, upgrade it, and then start it again. We don't really bother about the application state. That is handled outside of the application.

When upgrading databases, we do need to consider state, because a database contains comparatively little logic and structure, but contains much state.

In order to describe a database structure change, we issue a command to change the structure.

The database structures before and after a change is applied should be seen as different versions of the database. How do we keep track of database versions?

Liquibase is a database migration system that, at its core, uses a tried and tested method. There are many systems like this, usually at least one for every programming language. Liquibase is well-known in the Java world, and even in the Java world, there are several alternatives that work in a similar manner. Flyway is another example for the Java platform.

Generally, database migration systems employ some variant of the following method:

- Add a table to the database where a database version is stored.
- Keep track of database change commands and bunch them together in versioned changesets. In the case of Liquibase, these changes are stored in XML files. Flyway employs a slightly different method where the changesets are handled as separate SQL files or occasionally as separate Java classes for more complex transitions.
- When Liquibase is being asked to upgrade a database, it looks at the metadata table and determines which changesets to run in order to make the database up-to-date with the latest version.

As previously stated, many database version management systems work like this. They differ mostly in the way the changesets are stored and how they determine which changesets to run. They might be stored in an XML file, like in the case of Liquibase, or as a set of separate SQL files, as with Flyway. This later method is more common with homegrown systems and has some advantages. The Clojure ecosystem also has at least one similar database migration system of its own, called Migratus.