Technology Conversations

Continuous Deployment: Strategies

This article is part of the Continuous Integration, Delivery and Deployment series.

Previous article provided introduction to continuous deployment. In this one we'll continue where we left and explore different strategies to deploy software. The article is in no way an exhaustive list of ways to deploy applications but tries to provide few common ways that are in use today.

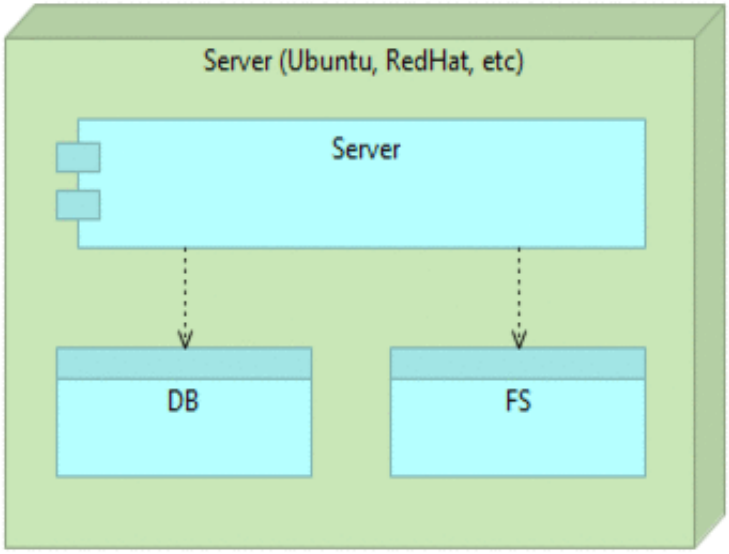
Mutable Monster Server

Common way to build and deploy applications is through **"mutable monster server"**. We create a web server that has the whole application and change every time there is a new release. Changes can be in configuration or code (JAR, WAR, static files, etc). Since we are changing it on every release, it is **mutable**.

With mutable servers we cannot know for certain that development, test and production environments are the same. Even different nodes in the production might have undesirable differences. Code, configuration or static files might not have been updated in all instances.

It is a **monster** server since it contains everything we need in a single instance. Back-end, front-end, APIs, etc. More over, it grows over time. It is not uncommon that after some time no one is sure what is the exact configuration of all pieces in production and the only way to accurately reproduce it somewhere else (new production node, test environment, etc) is to copy VM where it resides and start fiddling with configurations (i.e. IPs, host file, DB connections, etc). We just keep adding to it until we lose track of what it has.

Simplified image of such a server would be following.



It is a single big monster server that communicates with the database (DB) and file system (FS).

It looks simple but it usually isn't. By coupling everything into one place we are hiding complexity thus increasing the chance of discrepancies between different instances.

Time to restart such a server when it receives a new release can be considerable. During that time server is usually not operational. Downtime that new release provokes is a loss of money and trust. Today's business expects us to operate 24/7 without any downtime and it is not uncommon that release to production means night work of the team during which our services are not available.

Testing is also a problem. No matter how much we tested the release on development and test environments, first time it will be tried in production is when we deploy it and make it available not only to our testers but also to all our users.

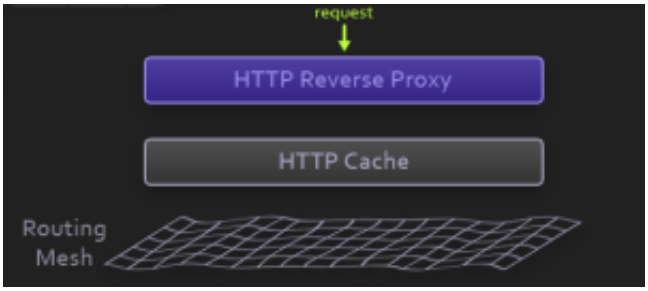
Moreover, fast rollback on such a server is close to impossible. Since it is mutable, there is no "photo" of the previous version.

By having architecture like this we cannot fulfill all, if any, of requirements described in the previous article. We cannot **deploy often** due to inability to produce **zero-downtime** and easily **rollback**. Full **automation** is risky due to mutable nature of its architecture thus preventing us to **be fast**.

By not deploying often we are accumulating changes that will be released and in that way we are increasing the probability of failure.

In order to solve those problems we should be immutable and composed of small, independent and self-sufficient applications. Remember, our goals are to deploy often, have zero-downtime, be able to rollback any release, be automated and be fast. More over, we should be able to test the release on production environment before actual users see it.

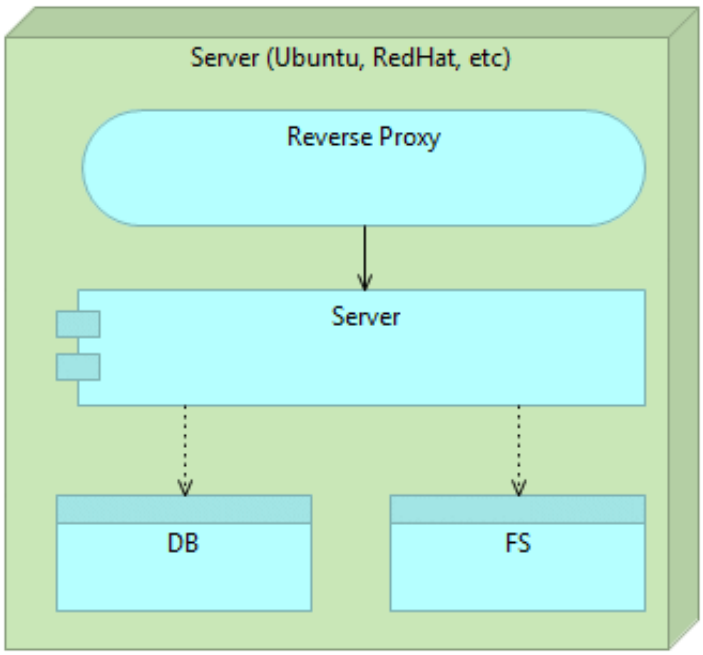
Immutable Server and Reverse Proxy



Each **"traditional"** deployment introduces a risk tied with changes that need to be performed on the server. If we change our architecture to immutable deployments, we gain immediate benefits. Provisioning of environments becomes much simpler since there is no need to think about applications (they are unchangeable). Whenever we deploy an image or container to the production server, we know that it is **exactly** the same as the one we built and tested. Immutable deployments reduce the risk tied to unknown. We know that each deployed instance is **exactly** the same as the other.

Reverse proxy can be used to accomplish zero-downtime. Immutable servers together with reverse proxy in a simplified form can be following.

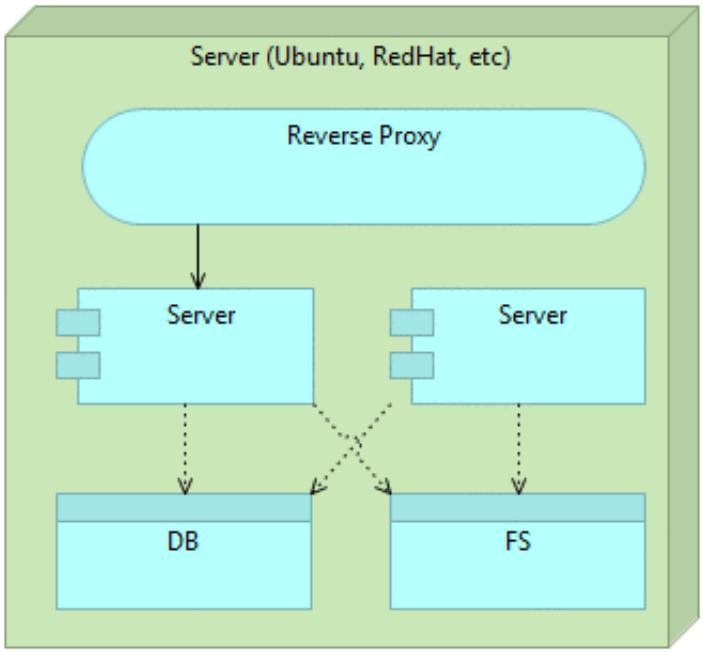
First we start with a reverse proxy that points to our server. There's nothing special about the proxy except that all traffic is routed through it instead of exposing the server directly.



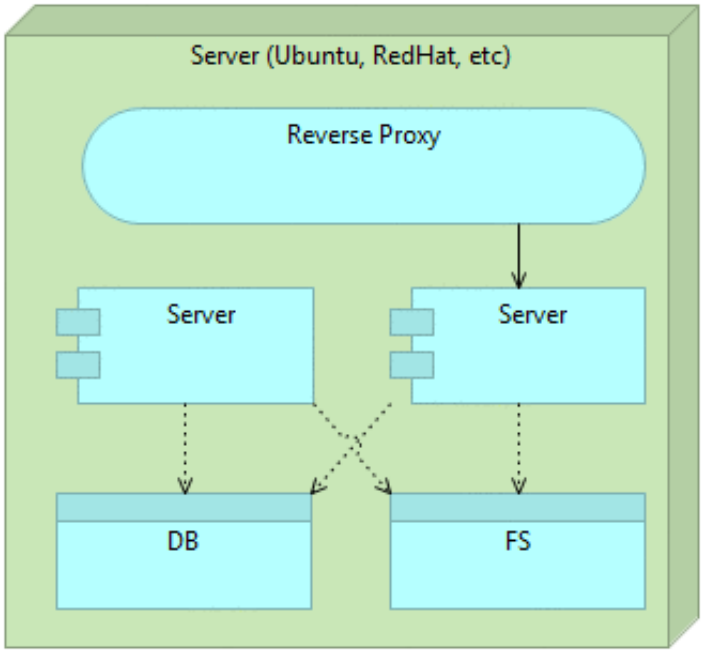
Once we decide to deploy a new version, we do it by deploying a separate instance of the server. At this moment we have two instances. One old (previous release) and one new (latest release). All traffic still goes to the old server through the **reverse proxy** so users of our application still do not notice any change. For them, we're still running the old and proven software. This is the good moment to run final set of tests. Preferably those tests are automatic and part of the deployment process but manual verification is not excluded. For example, if changes were done to front-end, we might want to do final round of user experience tests. No matter what types of tests are performed,

they should all “attack” the new release bypassing the reverse proxy. Good thing about those tests is that we are working with future production version of the software that resides on production hardware. We are testing production software and hardware without affecting our users (they are still being redirected to the old version). We could even enable our new release only to limited number of users in form of A/B testing.

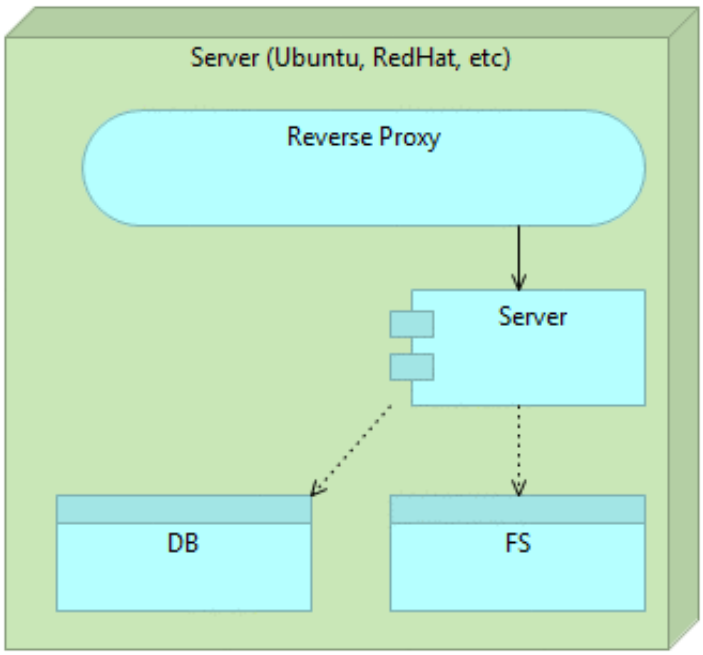
To summarize, at this stage we have 2 instances of the server, one (previous release) used by our users and the other (latest release) used for testing.



Once we are finished with tests and are confident that the new release works as expected, all we have to do is change the reverse proxy to point to the new release. The old one can stay for a while in case we need to rollback the changes. However, for our users it does not exist. All traffic is routed to the new release. Since the latest release was up-and-running before we changed the routing, switch itself will not interrupt our service (unlike, for example, if we would need to restart the server in case of mutable deployment). When route is changed we need to reload our reverse proxy. NGINX, for example, maintains old connections until all of them are switched to the new route.



Finally, when we do not need the old version, we can remove it. Even better, we can let the next release remove it for us. When the time comes, release process will remove the older release and start the process all over again.



The technique described above is called **blue-green deployment** and has been in use for a long time.

Immutable Microservices

We can do even better than this. With immutable deployments we can easily accomplish automatism of the process. Reverse proxy gives us zero-downtime and, having two releases up and

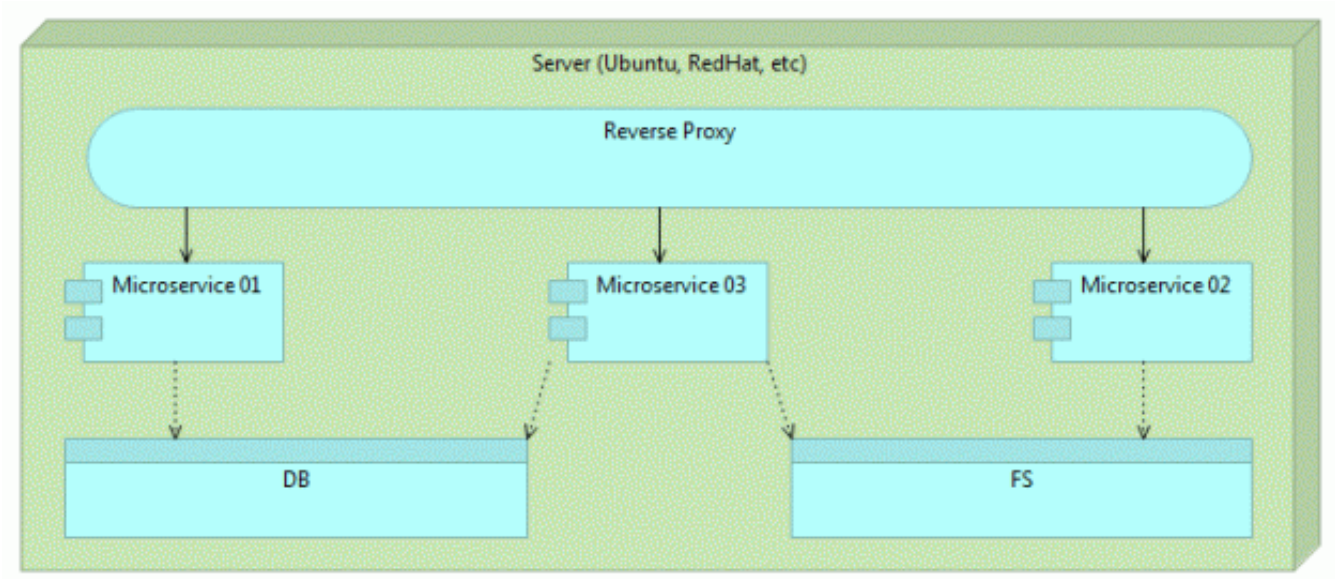
running allows us to rollback easily. However, since we're still dealing with one big server, deployment and tests might take a long time to run. That in itself might prevent us from being fast and thus from deploying as often as needed. Moreover, having everything as one big server increases development, testing and deployment complexity. If things could be split into smaller pieces, we might divide complexity into easily manageable chunks. As a bonus, having small independent services would allow us to scale more easily. They can be deployed to the same machine, scaled out across the network or multiplied if performance on one of them becomes bottleneck. Microservices to the rescue!

Microservices is the architectural approach to develop a single application as a set of small and, when possible, independent services. When one of our microservices gets updated, we can deploy it and leave the rest of the system intact.

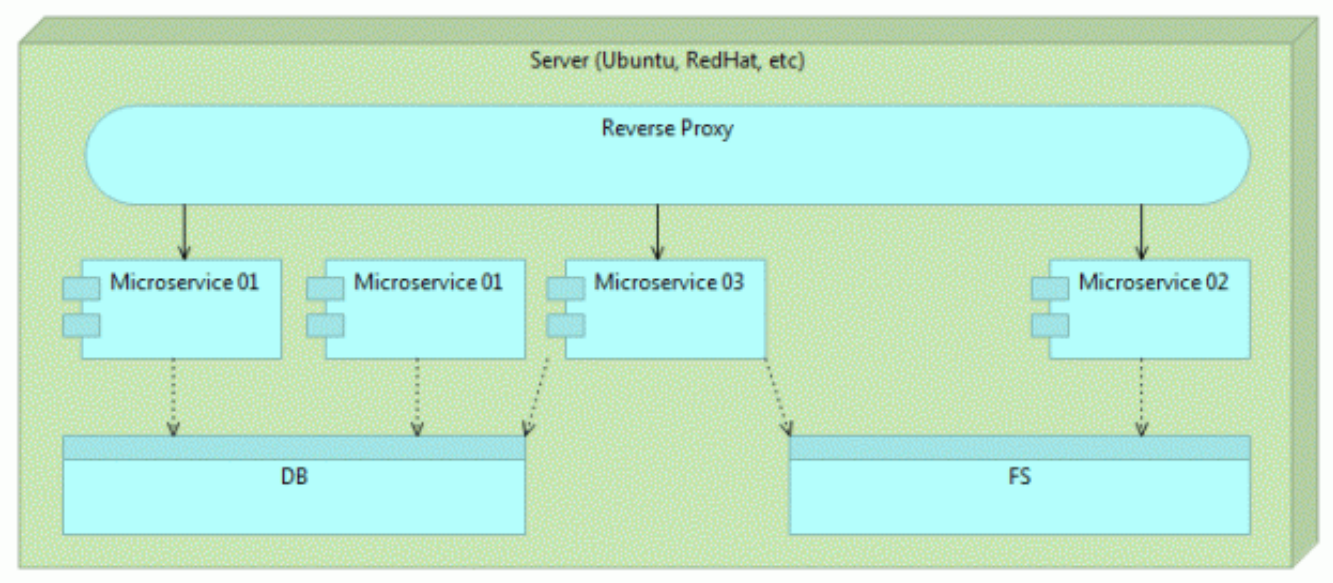
With **"monster application"** we tend to have decoupled layers. Front-end code should be separated from back-end, business layer from data access layer, etc. With microservices we should start thinking in a different direction. While back-end and front-end should be separated, back-end itself can be sliced vertically. Instead of having business layer separated from data access layer, we would separate services. For example, users management could split from sales service. Another difference is physical. While traditional architecture separates on a level of packages and classes but still deploys everything together, microservices are split physically. Previous example of users management and sales services would be developed in separate projects, deployed and tested as separate instances and processes. They might not even reside on the same physical machine.

Deployment of microservices follows the same pattern as previously described.

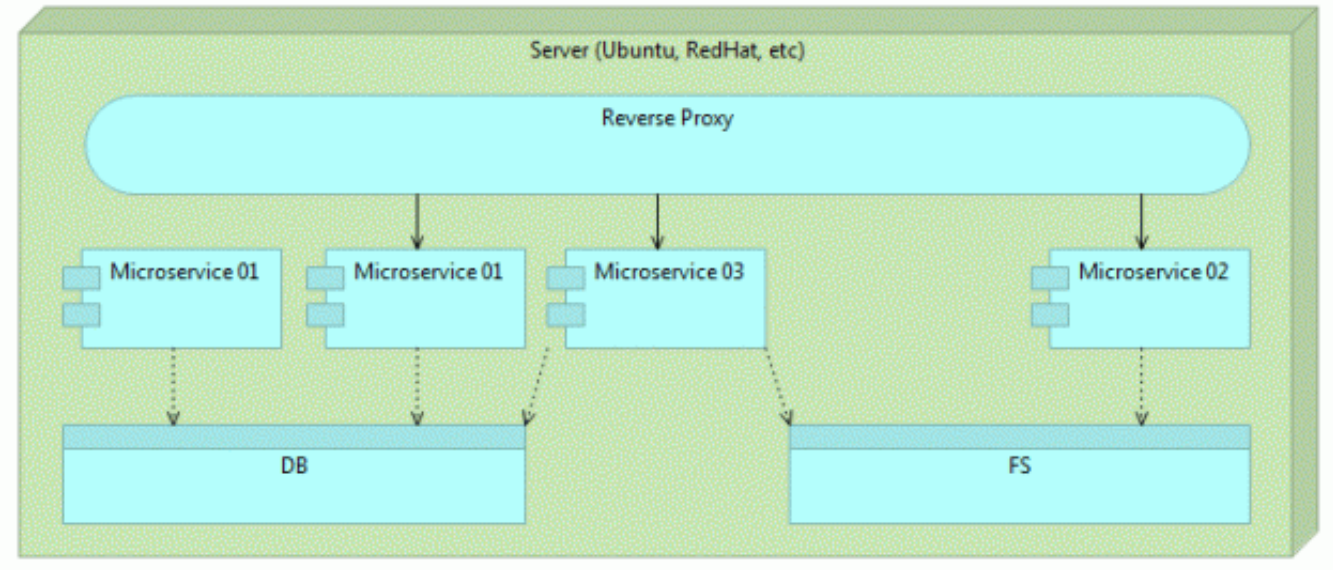
We deploy our microservice as any other software.



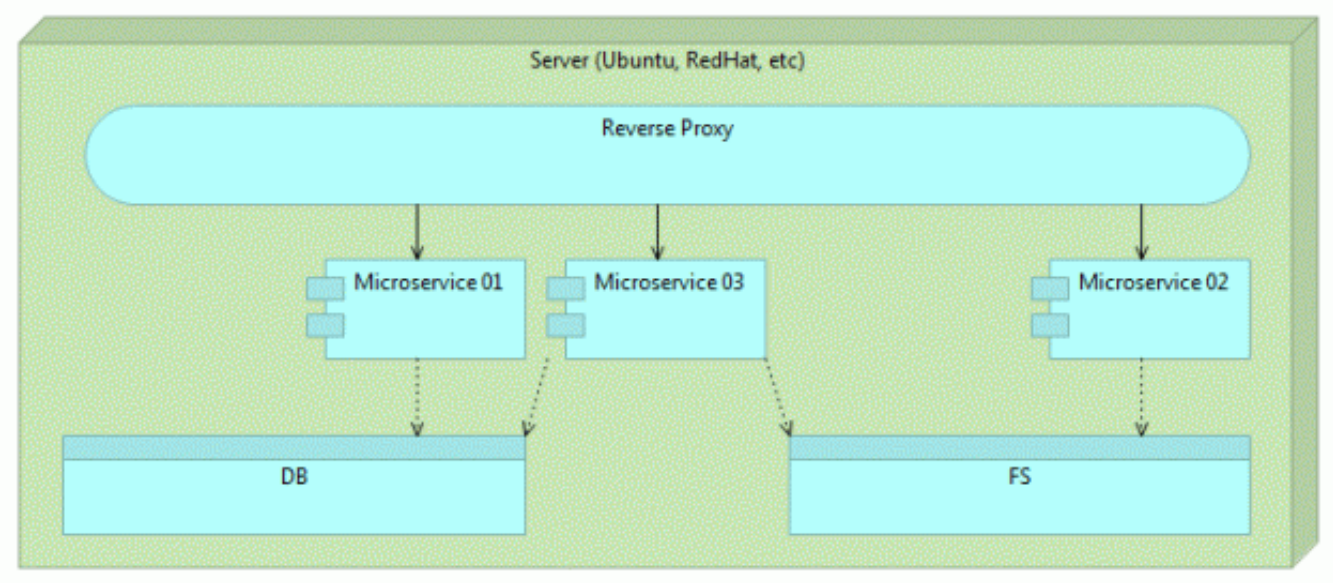
When the time comes to release a new version of some microservice we deploy it alongside the older version.



When that microservice release is properly tested we change the route.



Finally, we remove the older version of the microservice.



Now we can truly **deploy often automatically, be fast with zero-downtime and rollback** in case something goes wrong.

Technologically this architecture might pose certain problems and that will be the subject of the next article. We'll go through hands-on practical implementation of continuous deployment.

photo credit: twm1340 via photopin cc

photo credit: Fenng(dbanotes) via photopin cc

This entry was posted in Continuous Integration, Delivery and Deployment and tagged Continuous delivery, Continuous integration, Continuous Integration, Delivery and Deployment, Immutable deployment, Immutable server, Microservices on December 3, 2014 [http://technologyconversations.com/2014/12/03/continuous-deployment-strategies/] by Viktor Farcic.

4 thoughts on "Continuous Deployment: Strategies"

Pingback: 2p – Continuous Deployment: Strategies – Offeryour.com Blog

mjedynak1
January 3, 2015 at 10:58 pm

The idea is very nice, however my question is:
how do you implement it when your application is using the database and new release is introducing schema changes (which are incompatible with the old release)?

Viktor Farcic Post author
January 3, 2015 at 11:46 pm

Databases are probably the most problematic part of the CD. In my experience what looks like incompatible change can often be done in a different way thus maintaining compatibility. Every case is different. In some cases I created views that provided data in old format. In others there was no way around it and we had to sacrifice ability to rollback. In those cases CD was switched off temporarily.

I think that the key is in very short cycles that CD proposes. If commit is a result of only few hours of work, changes to the DB are usually small and can be made to be backwards compatible. Also, NoSQL makes this situation much easier.

Another think that worked well in some cases are feature toggles. Data that uses old schema continues using the old code and everything new goes through the new code. Once new code is proven to work data is transformed to the new schema and toggles (with the old code) removed.

In any case, it's hard to give a general advice. Every case is potentially different and we need to deal with new obstacles as they arise.

mjedynak1
January 5, 2015 at 8:18 pm

I suspected that there is no easy general solution. Thanks for the explanation.