LordElph's Ramblings

Stuff and nonsense about software development and whatever else I find fun…

Blue-Turquoise-Green Deployment

In this post I'm putting a name to something I've found myself doing in order to deliver zero-downtime deployments without any loss of database consistency.

The idea of Blue-Green deployment is well-established and appealing. Bring up an entire new stack when you want to deploy, and when you're ready, flip over to it at the load balancer.

Zero downtime deployment. It makes everyone happy.

But…data synchronization is hard

Cloud environments make it easy to bring up a new stack for blue-green deployments. What's not so easy is dealing with transactions during the flip from blue to green.

During that time, some of your blue services might be writing data into the blue database, and on a subsequent request, trying to read it out of green.

You either have to live with a little inconsistency, or drive yourself crazy trying to get it synchronized.

What about a common database?

This won't suit all applications, but you can do blue-green deployment with a common data storage backend. The actual blue and green elements of the stack consist of application code and any data migration upgrade/downgrade handling logic.

Most of the time, if you're trying to push out frequent updates, those updates are software changes with infrequent database schema changes.

So, you can happily make several releases a day with zero downtime. However, sooner or later you're going to make a breaking schema change.

The horror of backwards incompatible schema changes

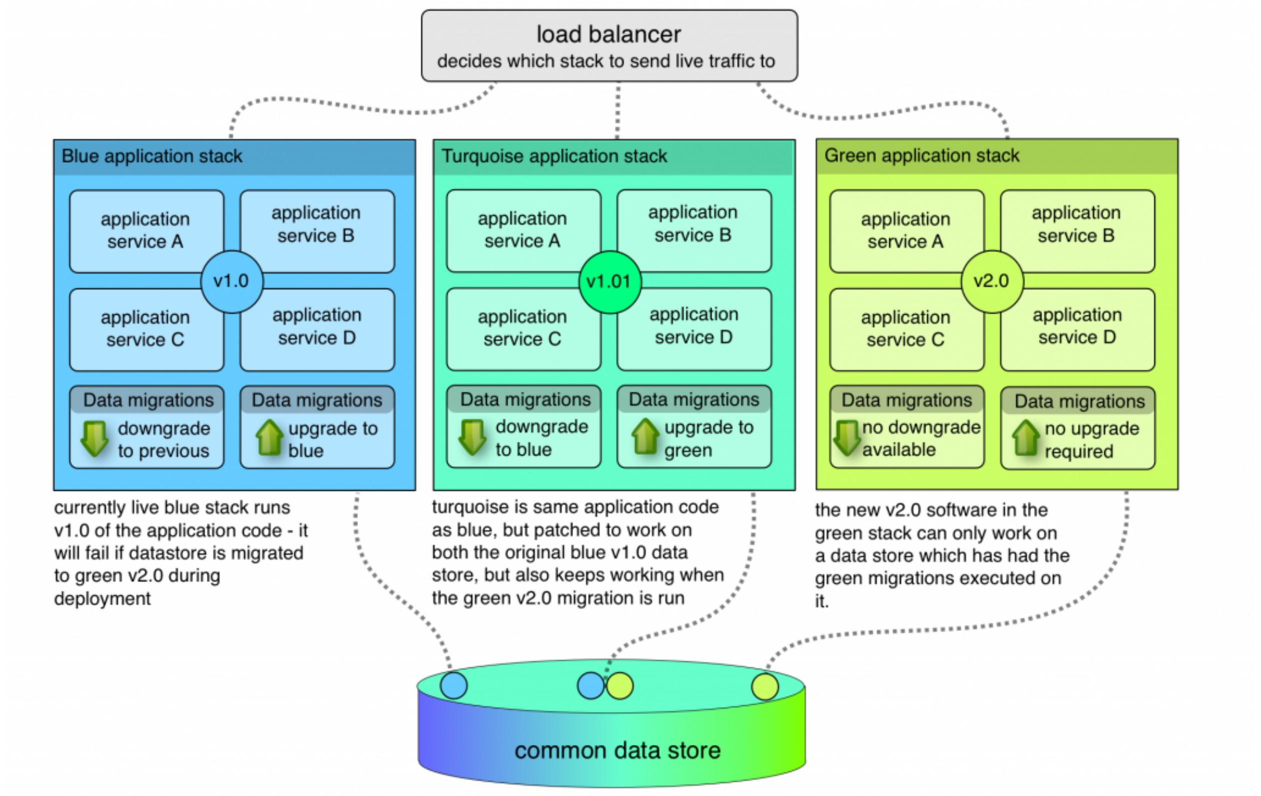
So, you're barrelling along with your shared data backend, and you find the current live blue deployment will fail when the new green deployment performs its database migrations on your common data store.

Now you can't deploy green without a scheduled downtime.

But you don't want scheduled downtime! How can we do a zero downtime deployment and still retain the green-blue rollback capability?

Introducing Blue-Turquoise-Green deployment!

You need to create a turquoise stack. That's the blue release, patched to run without failure on both a blue and a green database schema. This means it might have to detect the availability of certain features and adapt its behaviour at runtime. It might look ugly, but you're not planning to keep it for long.



Now, you can perform a deployment of turquoise. It runs just fine on the blue database, and you can run the database migrations for green. It keeps on trucking. Now you're safely running blue on a green-compatible database, you go ahead and deploy the green stack.

If you do run into problems, you've got everything in place to downgrade. Flip from green back to turquoise. Revert the database migrations, and you can then flip from turquoise to blue, and you're back where you started.

Thinking in turquoise

For me, this has been more of a thought experiment. I've found that if you plan to do blue-green deployment on a shared data backend, you naturally adopt a 'turquoise' mindset to the migrations.

That means ensuring you design schema changes carefully, and deploy them in advance of the code which actually requires them. In order words, you build in that turquoise-coloured forward compatibility ahead of time, and you're back to low risk, blue-green deployments!

Finally, why turquoise?

Because turquoise is a much nicer word than cyan. I should also say that I don't claim this is a new idea. Giving a name to things makes it easier to discuss with others – I was trying to describe this approach to someone and wrote this as a result. Comments are welcome.

This entry was posted in DevOps, Software Development on February 11, 2015 [http://blog.dixo.net/2015/02/blue-turquoise-green-deployment/] by Paul Dixon.

2 thoughts on "Blue-Turquoise-Green Deployment"

Greg
February 11, 2015 at 10:52 pm

What do you use for your DB migrations?

I've been looking at flywaydb but haven't used it in anger yet.

Paul Dixon Post author
February 11, 2015 at 11:13 pm

I use Doctrine Migrations – so far, that's been just fine. I haven't used flywaydb, but from what I can tell, it takes a similar approach to Doctrine. The main problem with this sort of automation is handling different code branches. When you merge them together, you need to carefully review the migrations to ensure you've got a sane upgrade/downgrade step.