

Migrating to Azure – Top Tips

InishTech migrated their entire SLPS software license management platform to Windows Azure in November 2010. Just 1 month on from the transition, InishTech Senior Azure Developer & Azure Project Lead Peter McEvoy gives some hints and tips for developers preparing for a similar project.

I guess our transition was modest enough, as we were already a SaaS provider. We initially installed the Azure SDK and Development fabric and started coding with that in Visual Studio 2008. A lot of our porting effort was to do with code that was using old frameworks that had nothing to do with Azure, so it was quite involved. We did need to upgrade to SQL Server 2008 R2 as part of the process, as that's the only client tool that can connect to SQL Azure at the moment.



We invested a lot of time and effort in the automation of key processes and procedures, and I think we very quickly saw the benefits of this decision. Here are my key takeaways looking back now, 1 month on....

Plan the Transition

It should go without saying - if you fail to plan, you should plan to fail. We planned this pretty carefully, and I am confident it was not wasted. Challenges we hit during the project and since going live have definitely been more manageable because we put so much time into the planning & preparation phase.

Don't just port it – Redevelop

Azure isn't just another platform. There's so much which your application can benefit from if you redevelop your application specifically for Azure. It may be more effort upfront, but I think probably less effort over time as you will have fewer issues with a clean build and you'll be able to take advantage of all the Azure-specific benefits.

Use the MSDN Azure Community

They are a great help and support. The Microsoft folk are also very quick, very responsive, and invariably any issues you have will get sorted by late afternoon when the guys from Redmond come online.

Setup a "Test" environment using a separate subscription

Azure has the notion of a "staging" slot for deployments, but these are not ideal for testing your application - settings used by the "staged" deployment are exactly the same as your production deployment so have the potential to interfere with your current running production deployment.

My advice would be to setup a separate Test environment that developers and QA can use to test the application. Preferably setup this environment under a separate subscription (login), as there is no way to set permissions within a subscription and there should be no chance that a test deployment can overwrite a production deployment.

We have set up automated scripts that will "tear down" all test and staging deployments after 19:00 each night thus reducing costs when those environments are not being used.

Automate Deployments with PowerShell commandlets

You should write scripts that will automate all parts of managing your deployments. At the very least, you should be able to:

- suspend an existing deployment;



PROTECT



LICENSE



MANAGE



MONETIZE

- delete a suspended deployment;
- deploy a new package (and associated configuration file);
- start the new deployment and then "swap" the running deployment into the production slot.

If you setup a Test environment (see previous point), then you will need to be able to manage multiple .cscfg configuration files. Using a script for deployment means that you can reduce the errors inherent with managing multiple environments.

There are a set of PowerShell commandlets available that can serve as the basis for a richer script - but they are not enough on their own.

Get Native Azure diagnostic logging working as soon as possible and get on top of "log shipping"

Initially you may feel that getting the native logging working is complicated. But you need to work it out and get it working: Once your application is deployed, it's the only way that you can find out what is going on when something goes wrong.

Azure uses a notion of "log shipping" which means that application generated log messages only get shipped to persistent storage (where you can read them) at most once per minute (1 minute is the shortest configurable interval). This is a little frustrating if you are trying to diagnose an issue.

When configuring logging, capture all event levels - even your application debug messages as they can always be filtered later.

Investing in a log reading tool like Cerebrata's "Azure Diagnostic Manager" is a good idea - it's cheaper than wasting your time writing your own, and more flexible than the native visual studio tool.

You should decide on a logging strategy within your application. Make sure warnings are *actually* warnings and that errors are errors. There is nothing worse than your logs being flooded with messages at the wrong level. At the very least, you should make sure you are logging enough information to help you diagnose an issue.

Learn about dependency injection and use it to separate Azure specific behaviour from IIS behaviour

Being able to continue to deploy and host your application in IIS is a huge benefit for agile development.

The dev fabric is great for isolating some Azure specific issues, but it cannot be used on build servers or automated test environments.

A "live" test Azure environment also cannot be used for automated build or testing chiefly because it can sometimes take 15 or 20 minutes to deploy and run an Azure deployment during which time you need to use the Azure portal to monitor when your application is "ready" (the PowerShell commandlets return a status of "running" even when your application is "initializing"!)

Thus, being able to continue to deploy to IIS can greatly help with application development: You can deploy your web application to a test server and run automated tests without the Azure deployment uncertainty.

Using Dependency Injection to isolate Azure specific behavior from your app means that you can inject IIS or Azure behavior at runtime. An example of Azure specific vs. IIS specific behavior would be the way that app settings are read from the configuration file. In IIS you want to use ConfigurationManager to read from the web.config, but in Azure you want to use RoleEnvironment.



Use Windows Scheduler to drive SQL Azure "jobs"

One of the reasons that the native SQL Session State Provider has not been fully ported to Azure is because there is no SQL "agent" that can run a job every 5 minutes to "Clear Expired Sessions". But, this kind of job can be driven externally by a simple on site windows scheduler script that runs "sqlcmd" against your DB instance. The same can be done to execute the "CREATE DATABASE ... AS COPY OF ..." command to backup your database

On site windows scheduler can also be used to drive other jobs such as a script to remove any deployments in your staging slots or test environments.

In Conclusion

That's all I can think of for the moment. Drop me a line if you've any questions, I'm always happy to offer an opinion!

pmcevoy@inistech.com