Chapter 1: Life before Vagrant

Manual Development Environments

- …are time consuming and therefore costly.
- …are a bad experience for your developers.
- …are rarely accurately reconstructed - it’s a game of Chinese whispers which produces imperfect clones.
- …encourages *manual configuration* rather than the more reliable and repeatable practice of *Infrastructure-as-code*. 
Dev vs Production Parity (or, “It works on my box!”)

- Minor differences in environment configuration can produce obscure and hard to find bugs in Production.
- …major differences can result in major ones!
- It is not uncommon to see development occurring on different operation systems, different versions of Windows Server and different versions of SQL Server. Production is only one combination of these.
- Ideally, we want all of our environments to be identical so that we are confident code shipped from one into another will operate as expected.
12 factor apps

We want to be able to:

● Use declarative formats for setup automation, to minimize time and cost for new developers joining the project.
● Have, a clean contract with the underlying operating system, offering maximum portability between execution environments.
● Design our apps for deployment on modern cloud platforms, obviating the need for servers and systems administration.
● Minimize divergence between development and production, enabling continuous deployment for maximum agility.
● Scale up without significant changes to tooling, architecture, or development practices.
● These principles are what make up The 12 factor app (12factor.net).

Vagrant can help us meeting factor #10: Dev/prod parity - “Keep development, staging, and production as similar as possible”.

Chapter 2: Vagrant & friends

What is Vagrant?

● A tool to help create and configure lightweight, reproducible, and portable development environments.
● A replacement for manual configuration and encourages Infrastructure-as-code.
● Typically, it will run a virtual machine of sorts (Virtualbox, VMWare, Parallels etc).
● Vagrant by default will share and keep in sync the current working directory (normally the root of your project) into a directory in the running VM.
● This allows you to author locally and run your code remotely, for example, in an application server.
● Vagrant configurations are just a Ruby file with a declarative syntax (DSL). Procedural-style code is also permitted for advanced or more complicated setups.
Vagrant has spun up Virtualbox VM with the name ‘Web’, and is running it on a Mac OSX desktop.

- Code is synchronised from /c/proj/x -> /vagrant of the running environment.
- The Web application spins up a server using the code in /vagrant, which binds to a private network on the VM on port 80.
- Port 8080 is exposed to the outside ‘world’ (in this case, a Mac OSX desktop) which forwards requests to port 80 in the running VM.
- The Developer is free to author in whichever IDE of her choosing.

The TAO of Vagrant ([http://mitchellh.com/the-tao-of-vagrant](http://mitchellh.com/the-tao-of-vagrant))

- Vagrant is designed with a specific goal: to make development easier and environments more reliable.
- Developers should be able to checkout their project and run `vagrant up` and have a fully working dev execution environment.
- Developers can still use their preferred development authoring environment - Visual Studio, Sublime, ViM, their terminal, their shortcuts - etc.
- Your authoring environment is sacrosanct and messing with it is not something Vagrant is intended to do.
- In an ideal world, the existence of Vagrant is transparent.
- If there is a new version of the environment or it has been ‘compromised’ with manual configuration `vagrant destroy` & `vagrant up` will destroy the machine and create a new, clean, isolated environment.
- This workflow is repeatable and transferrable to other projects.
Chapter 3: Developing Ecosystems

From Monolith to Micro-services

- Applications are usually more interesting than a single UI, they are usually connected to things; part of a functioning ecosystem.
- A well designed application expects backing services to be provided as attached resources (Factor #4 of 12factor.net).
- Whilst Vagrant provides a good solution to large Monoliths (i.e. Everything on one machine) it is also a good fit to build ecosystems.
- Vagrant allows us to create these ecosystems locally, while still having the benefits of isolation, repeatability, immutability etc.
- Running ecosystems locally gives us enormous amount of flexibility, allowing us to simulate or test a number of scenarios, for example to:
  - Model distributed systems behaviour
  - Perform disaster testing; outages, network partitions etc.
  - Replace components independently
  - Test OS/DB upgrades
  - Run entire production replica ecosystems, locally

Potential Talent Search Architecture
What about Docker?

- Vagrant augments Docker, it doesn’t replace it.
- Vagrant can run ecosystems using Docker under the hood; in fact it’s also a good environment to iterate on Docker images and provision them.
- Docker is a good container choice for ecosystems due to its lightweight footprint.

The following is a Logical View of a fictional ecosystem with Web and API applications exposed to the ‘World’ and a database and Hadoop application hidden ‘behind the firewall’:

- Similar to the previous example, we expose multiple ports to the outside world which map (via NAT) into 2 different running VMs.
- The 4 VMs are able to communicate through a private network on their native ports (80, 80, 27017 and 50070 respectively).
Chapter 4: DSC / Configuration Management Tools

What is DSC?

- Stands for “Desired State Configuration” - it is a tool for automating the configuration of servers, following the principle of Infrastructure-as-code.
- Replaces ad-hoc Powershell scripts or manual administration, with a standardised approach to managing systems.
- It was created by the Powershell Team and is used heavily at StackOverflow, and also internally for Dev Machine Automation and the Notifications project.
- A declarative configuration management tool, similar to Puppet & Chef, but is well suited to Windows as it uses native PowerShell under the hood.
- Can be run in an ‘ad hoc’ (push) or ‘centrally managed’ (pull) approach to be keep the server in a desired state to prevent ‘configuration drift’.
- Designed for idempotency. Can be run many times without breaking things.
- Requires WMF 5.0+ (Poweshell 4+) to run.

Core Concepts

- Configurations: The main ‘run sheet’ of the program.
- Coordinates and collects all of the things that need to be done.
- Resources: The general definition (recipe) that enables the configuration of thing (e.g. a Website, a File or a Windows Feature).
- Configuration Data: Parameterises the Configuration for reusability.
- Composite Resources: Reusable Configurations that can be shared and used in other Configuration files (e.g. Web Server configuration, Package installs).

Example DSC Resource
Example DSC Configuration File

Using DSC

- DSC is fairly easy to learn if you start simple and build complexity.
- Use Vagrant in conjunction with the Vagrant DSC Plugin ([https://github.com/mefellows/vagrant-dsc/](https://github.com/mefellows/vagrant-dsc/)) to iterate on a machine and get started quickly.
- Cattle vs Pets - DSC is perfect for provisioning immutable (Windows) servers such as creating AMIs or local Vagrant boxes (Cattle), but it is also well suited to managing fleets of servers (Pets) when the immutable servers approach can’t be applied (e.g. legacy environments, databases, email servers etc).
- It’s still immature, error handling/messages/debugging can be rather tedious.
Chapter 5: Machine image pipelines with Packer

What is Packer and why might we want it?

- DSC and Vagrant only solve part of the equation; DSC requires a server and someone to kick it off and Vagrant needs a machine to boot - who creates that machine in the first place?
- Chef and other Configuration Management tools also suffer the same problem.
- How do we ensure other environments are configured the same as Dev with those same DSC resources? We want environment parity.
- Packer is "a tool for creating identical machine images for multiple platforms from a single source configuration" and can solve these problems for us.
- Packer will boot a machine from an image such as an ISO or an AMI, provision / configure it with scripts, DSC, Puppet etc. and then export it into multiple formats - such as a Vagrant file or an AMI.
- It can do this in parallel and can also take output from itself to iterate on a box.
- Using Packer, we can then create a ‘Machine Factory’ (aka Machine Image pipelines).
- Packer can bring an additional quality control step into the whole process by ensuring environments are consistent before coding even begins.

Using Packer

- DSC and Packer are a good combination and can be used to create pre-provisioned images.
- Image creation with CI is possible, although creating virtual machines within virtual machines can be problematic (this is not a problem for Docker images).
- Packer itself doesn’t support Windows well OOTB, as it requires the machine being provisioned to have SSH installed, which is poorly implemented and broken when installed on Windows.
- Install the Packer Community plugins project (https://github.com/packer-community/packer-windows-plugins) instead which replaces the SSH communicator with a native and reliable Windows implementation (using WinRM)

Packer and AWS

A typical workflow would involve:

1. Creating a base image with all Windows updates and company policies/modules applied (e.g. AV, licenses or patch levels).
2. Generating 2 Packer images with all of the application dependencies (e.g. IIS with Websites configured, firewall rules and so on) - one Vagrant box for development and an AMI for stage/prod.
3. Repeat steps 1 & 2 for all of the machine images required in the ecosystem.
4. Use Cloud-formation and/or a deployment script to spin up the newly created AMIs in a stack.
5. Inject environment variables into the server at stack creation time and push the code onto it.
6. Promote the new environment & discard the previous running application servers after traffic has drained in a blue/green style deployment.

Chapter 6: Practical Tips & Lessons from the field

Vagrant

- Avoid other development environment dependencies - Ideally Vagrant, a VM provider and plugins are all that are required to run your dev environment
- Virtualbox shared folders and IIS
  - Use rsync instead or network shares instead
- When provisioning scripts are too big/slow, move into a Packer script and bake into the base image - `vagrant up` should be fast.
- Always set `VAGRANT_HOME` to a large drive on multi-partitioned machines due to disk space restrictions (n+1).
- Distributing boxes
  - Upload to Atlas (atlas.hashipcorp.com) with a version number where possible
  - S3 or network share as a backup, using the `config.vm.box_url` setting to make discovering and downloading images easier.
- GUI Testing (e.g. Protractor tests) & debugging can be tough on headless environments
  - Consider having a GUI fallback (at the cost of slightly bigger images).
  - If GUI not feasible, you can forward the X display `X-Forwarding` on *nix hosts.
- Cross-OS concerns do occasionally come into play
  - NPM for example has issues writing to a linux networked file system where the underlying host is Windows.
- Windows Domain access
  - Try to avoid where possible.
If you can’t avoid, you can sysprep the machine with Packer and use `config.vm.hostname` and pass in a variable to dynamically set the domain during boot.

**DSC**

- Expect pain/effort, it’s in its infancy and is immature. In particular, error messages can be very unhelpful/confusing, and you should be prepared to get comfortable with Windows debugging in the event viewer and the CLI.
- Even so IMHO, DSC is still a better choice than the *nix open source alternatives.
- Use the Vagrant and the DSC Plugin ([https://github.com/mefellows/vagrant-dsc/](https://github.com/mefellows/vagrant-dsc/)) to iterate on a box; Vagrant allows you to test your DSC configurations in a clean environment and test any potential issues with conditions/ordering, without polluting your host environment.
- Avoid the Script resource - write your own custom resource instead.

**Packer**

- When building a new image, create scripts, run them manually and then automate once you’re comfortable they’re working as expected.
- Iteration times can be slow, particularly when booting and installing an OS.
- Using Vagrant to test those scripts is often a good starting point.
- Don’t try to do it all in one go. Build complexity.
- Build Packer images in pipelines - Feed output of Packer runs into the start of other Packer runs:
  - For example, use the Virtualbox ISO builder to boot and install a Windows machine with updates applied.
  - That box can now be fed into the Virtualbox OVF builder which runs the application specific provisioners (typically this is the more error prone step).
- If you’re creating a Windows image, Use Packer Community (only if you can’t use Linux! :)).
- Don’t forget to disable Windows Updates (at least for dev machines).
- Sysprep Windows machines so licenses don’t expire.
Resources

- Vagrant (http://vagrantup.com)
- The TAO of Vagrant (http://mitchellh.com/the-tao-of-vagrant)
- Vagrant DSC Plugin (https://github.com/mefellows/vagrant-dsc/)
- Read the free DSC eBook (https://onedrive.live.com/?cid=7f868aa697b937fe&id=7F868AA697B937FE!107) for a small but comprehensive guide to DSC
- SEEK-Jobs/DSC resource (https://github.com/SEEK-Jobs/DSC/)
- Setting up a server for use with DSC (see packer templates https://github.com/mefellows/packer-community-templates)
- Packer (http://packer.io)
- Sample Windows Packer Templates (https://github.com/mefellows/packer-community-templates)
- Legacy Windows Packer Templates and provisioning scripts (Uses OpenSSH, however there is plenty of inspiration to be found) - https://github.com/joefitzgerald/packer-windows)