Sample Chapter

Azure IN ACTION

Chris Hay
Brian H. Prince

MANNING

Azure in Action

by Chris Hay and Brian H. Prince

Chapter 3

brief contents

Part 2

Understanding the Azure service model

With the cloud basics and Windows Azure concepts under your belt, we dial it up a notch. In part 2, we look at all the parts of the service model.

Chapter 3 explains what the service model is, how Azure uses it, and how Azure works behind the scenes. A brilliant chapter if there ever was one.

The quality only gets better as we move into chapter 4, which discusses how to reference the Azure APIs in your code and how to exploit the service runtime.

In chapter 5, we trot out how to configure your service model using the configuration files and the portal. An exciting chapter, especially if you like XML and angle braces.

3 How Windows Azure works

This chapter covers

- How Microsoft built Azure
- What a cloud operating system is
- How your application is provisioned and managed in the cloud

Now that you have a basic understanding of what you can do with Azure, let's drill deeper into the pieces of Azure and how to best work with them. In this chapter, we'll discuss how Windows Azure is architected and how it does the cloud magic that it does. Understanding this background will help you develop better services, be a better person, and get the most out of your Azure infrastructure.

3.1 The big shift

When Azure was first announced at the PDC in 2008, Microsoft wasn't a recognized player in the cloud industry. It was the underdog to the giants Google and Amazon, which had been offering cloud services for years by that time. Building and

deploying Azure was a big bet for Microsoft. It was a major change in the company's direction, from where Microsoft had been and where it needed to go in the future. Up until that time, Microsoft had been a product company. It designed and built a product, burnt it to CD, and sold it to customers. Over time, the product was enhanced, but the product was installed and operated in the client's environment. The trick was to build the right product at the right time, for the right market.

With the addition of Ray Ozzie to the Microsoft culture, there was a giant shift toward services. Microsoft wasn't abandoning the selling of products, but it was expanding its expertise and portfolio to offer its products as services. Every product team at Microsoft was asked if what they were doing could be enhanced and extended with services. They wanted to do much more than just put Exchange in a data center and rent it to customers. This became a fundamental shift in how Microsoft developed code, how the code was shipped, and how it was marketed and sold to customers.

This shift toward services wasn't an executive whim, thought up during an exclusive executive retreat at a resort we'll never be able to afford to even drive by. It was based on the trends and patterns the leaders saw in the market, in the needs of their customers, and on the continuing impact of the internet on our world. Those in charge saw that people needed to use their resources in a more flexible way, more flexible than even the advances in virtualization were providing. Companies needed to easily respond to a product's sudden popularity as social networking spread the word. Modern businesses were screaming that six months was too long to wait for an upgrade to their infrastructure; they needed it now.

Customers were also becoming more sensitive to the massive power consumption and heat that was generated by their data centers. Power and cooling bills were often the largest component of their total data-center cost. Coupling this with a concern over global warming, customers were starting to talk about the greening of IT. They wanted to reduce the carbon footprint that these beasts produced. Not only did they want to reduce the power and cooling waste, but also the waste of lead, packing materials, and the massive piles of soda cans produced by the huge number of server administrators that they had to employ.

3.1.1 The data centers of yore

Microsoft is continually improving all the important aspects of its data centers. It closely manages all the costs of a data center, including power, cooling, staff, local laws, risk of disaster, availability of natural resources, and many other factors. While managing all this, it has designed its fourth generation of data centers. Microsoft didn't just show up at this party; it planned it by building on a deep expertise in building and running global data centers over the past few decades.

The first generation of data centers is still the most common in the world. Think of the special room with servers in it. It has racks, cable ladders, raised floors, cooling, uninterruptable power supplies (UPSs), maybe a backup generator, and it's cooled to a temperature that could safely house raw beef. The focus is placed on making sure

the servers are running; no thought or concern is given to the operating costs of the data center. These data centers are built to optimize the capital cost of building them, with little thought given to costs accrued beyond the day the center opens. (By the way, the collection of servers under your desk doesn't qualify as a Generation 1 data center. Please be careful not to kick a cord loose while you do your work.)

Generation 2 data centers take all the knowledge learned by running Generation 1 data centers and apply a healthy dose of thinking about what happens on the second day of operation. Ongoing operational costs are reduced by optimizing for sustainability and energy efficiency. To meet these goals, Microsoft powers its Quincy, Washington, data center with clean hydroelectric power. Its data center in San Antonio, Texas, uses recycled civic gray water to cool the data center, reducing the stress on the water sources and infrastructure in the area.

3.1.2 The latest Azure data centers

Even with the advances found in Generation 2 data centers, companies couldn't find the efficiencies and scale needed to combat rising facility costs, let alone meet the demands that the cloud would generate. The density of the data center needed to go up dramatically, and the costs of operations had to plummet. The first Generation 3 data center, located in Chicago, Illinois, went online on June 20, 2009. Microsoft considers it to be a mega data center, which is a class designation that defines how large the data center is. The Chicago data center looks like a large parking deck, with parking spaces and ramps for tractor trailers. Servers are placed into containers, called *CBlox*, which are parked in this structure. A smaller building that looks more like a traditional data center is also part of the complex. This area is for high-maintenance workloads that can't run in Azure.

CBlox are made out of the shipping containers that you see on ocean-going vessels and on eighteen wheelers on the highways. They're sturdily built and follow a standard size and shape that are easy to move around. One CBlox can hold anywhere from 1,800 to 2,500 servers. This is a massive increase in data-center density, 10 times more dense than a traditional data center. The Chicago mega data center holds about 360,000 servers and is the only primary consumer of a dedicated nuclear power plant core run by Chicago Power & Light. How many of your data centers are nuclear powered?

Each parking spot in the data center is anchored by a refrigerator-size device that acts as the primary interconnect to the rest of the data center. Microsoft developed a standard coupler that provides power, cooling, and network access to the container. Using this interconnect and the super-dense containers, massive amounts of capacity can be added in a matter of hours. Compare how long it would take your company to plan, order, deploy, and configure 2,500 servers. It would take at least a year, and a lot of people, not to mention how long it would take to recycle all the cardboard and extra parts you always seem to have after racking a server. Microsoft's goal with this strategy is to make it as cheap and easy as possible to expand capacity as demand increases.

The containers are built to Microsoft's specifications by a vendor and delivered on site, ready for burn-in tests and allocation into the fabric. Each container includes networking gear, cooling infrastructure, servers, and racks, and is sealed against the weather.

Not only are the servers now packaged and deployed in containers, but the necessary generators and cooling machinery are designed to be modular as well. To set up an edge data center, one that's located close to a large-demand population, all that's needed is the power and network connections, and a level paved surface. The trucks with the power and cooling equipment show up first, and the equipment is deployed. Then the trucks with the computing containers back in and drop their trailers, leaving the containers on the wheels that were used to deliver them. The facility is protected by a secure wall and doorway with monitoring equipment. The use of laser fences is pure speculation and just a rumor, as far as we know. The perimeter security is important, because the edge data center doesn't have a roof! Yes, no roof! Not using a roof reduces the construction time and the cooling costs. A roof isn't needed because the containers are completely sealed.

Microsoft opened a second mega data center, the first outside the United States, in Dublin, Ireland, on July 1, 2009. When Azure became commercially available in January 2010, the following locations were known to have an Azure data center: Texas, Chicago, Ireland, Amsterdam, Singapore, and Hong Kong. Although Microsoft won't tell where all its data centers are for security reasons, it purports to have more than 10 and fewer than 100 data centers. Microsoft already has data centers all over the world to support its existing services, such as Virtual Earth, Bing Search, Xbox Live, and others. If we assume there are only 10, and each one is as big as Chicago, then Microsoft needs to manage 3.5 million servers as part of Azure. That's a lot of work.

3.1.3 How many administrators do you need?

Data centers are staffed with IT pros to care and feed the servers. Data centers need a lot of attention, ranging from hardware maintenance to backup, disaster recovery, and monitoring. Think of your company. How many people are allocated to manage your servers? Depending on how optimized your IT center is, the ratio of person-to-servers can be anywhere from 1:10 to 1:100. With that ratio, Microsoft would need 35,000 server managers. Hiring that many server administrators would be hard, considering that Microsoft employs roughly 95,000 people already.

To address this demand, Azure was designed to use as much automation as possible, using a strategy called *lights-out operations*. This strategy seeks to centralize and automate as much of the work as possible by reducing complexity and variability. The result is a person-to-servers ratio closer to 1:30,000 or higher.

Microsoft is achieving this level of automation mostly by using its own off-the-shelf software. Microsoft is literally eating its own dog food. It's using System Center Operations Manager and all the related products to oversee and automate the management of the underlying machines. It's built custom automation scripts and profiles, much like any customer would do.

One key strategy in effectively managing a massive number of servers is to provision them with identical hardware. In traditional data centers where we've worked, each year brought the latest and greatest of server technology, resulting in a wide variety of technology and hardware diversity. We even gave each server a distinct name, such as Protoss, Patty, and Zelda. With this many servers, you can't name them; you have to number them. Not just by server, but by rack, room, and facility. Diversity is usually a great thing, but not when you're managing millions of boxes.

The hardware in each Azure server is optimized for power, cost, density, and management. The optimization process drives exactly which motherboard, chipset, and every other component needs to be in the server; this is truly bang for your buck in action. Then that server recipe is kept for a specific lifecycle, only moving to a new bill of materials when there are significant advantages to doing so.

3.1.4 Data center: the next generation

Microsoft isn't done. It's already spent years planning the fourth generation of data centers. Much like the edge data center we described previously, the whole data center is located outside. The containers make it easy to scale out the computing resources as demand increases; prior generations of data centers had to have the complete data center shell built and provisioned, which meant provisioning the cooling and power systems as if the data center were at maximum capacity from day one. The older systems were too expensive to expand dynamically. The fourth generation data centers are using an extendable spine of infrastructure that the computing containers need, so that both the infrastructure and the computing resources are easily scaled out (see figure 3.1). All of this is outside, in a field of grass, without a roof. They'll be the only data centers in the world that need a grounds crew.

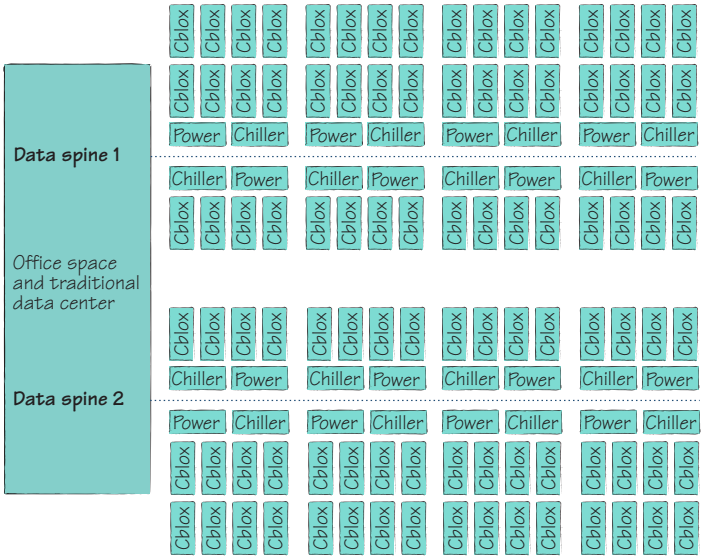

Figure 3.1 Generation 4 data centers are built on extensible spines. This configuration makes it easy to add not only computational capacity, but the required infrastructure as well, including power and cooling.

OK, you're impressed. Microsoft has a lot of servers, some of them are even outside, and all the servers are managed in an effective way. But how does the cloud really work?

3.2 Windows Azure, an operating system for the cloud

Think of the computer on your desk today. When you write code for that computer, you don't have to worry about which sound card it uses, which type of printer it's connected to, or which or how many monitors are used for the display. You don't worry, to a degree, about the CPU, about memory, or even about how storage is provided (solid-state drive [SSD], carrier pigeon, or hard disk drive). The operating system on that computer provides a layer of abstraction away from all of those gritty details, frees you up to focus on the application you need to write, and makes it easy to consume the resources you need. The desktop operating system protects you from the details of the hardware, allocates time on the CPU to the code that's running, makes sure that code is allowed to run, plays traffic cop by controlling shared access to resources, and generally holds everything together.

Now think of that enterprise application you want to deploy. You need a DNS, networking, shared storage, load balancers, plenty of servers to handle load, a way to control access and permissions in the system, and plenty of other moving parts. Modern systems can get complicated. Dealing with all of that complexity by hand is like compiling your own video driver; it doesn't provide any value to the business. Windows Azure does all this work, but on a much grander scale and for distributed applications (see figure 3.2) by using something called the *fabric*. Let's look into this fabric and see how it works.

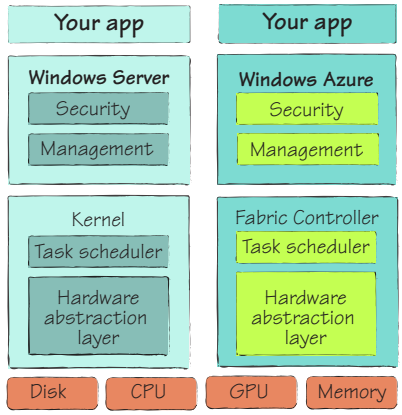

Figure 3.2 The Fabric Controller is like the kernel of your desktop operating system. It's responsible for many of the same tasks, including resource sharing, code security, and management.

Windows Azure takes care of the whole platform so you can focus on your application. The term *fabric* is used because of the similarity of the Azure fabric to a woven blanket. Each thread on its own is weak and can't do a lot. When they're woven together into a fabric, the whole blanket becomes strong and warm. The Azure fabric consists of thousands of servers, woven together and working as a cohesive unit. In Azure, you don't need to worry about which hardware, which node, what underlying operating system, or even how the nodes are load balanced or clustered. Those are just gritty details best left to someone else. You just need to worry about your application and whether it's operating effectively. How much time do you spend wrangling with these details for your on-premises projects? It's probably at least 10–20 percent of the total project cost in meetings alone. There are savings to be gained by abstracting away these issues.

In fact, Azure manages much more than just servers. There are plenty of other assets that are managed. Azure manages routers, switches, IP addresses, DNS servers, load balancers, and dynamic virtual local area networks (VLANs). In a static data center, managing all these assets is a complex undertaking. It's even more complex when you're managing multiple data centers that need to operate as one cohesive pool of resources, in a dynamic and real-time way.

If the fabric is the operating system, then the *Fabric Controller* is the kernel.

3.3 The Fabric Controller

Operating systems have at their core a kernel. This kernel is responsible for being the traffic cop in the system. It manages the sharing of resources, schedules the use of precious assets (CPU time), allocates work streams as appropriate, and keeps an eye on security. The fabric has a kernel called the Fabric Controller (FC). Figure 3.3 shows the relationship between Azure, the fabric, and the FC. Understanding these relationships will help you get the most out of the platform.

The FC handles all of the jobs a normal operating system's kernel would handle. It manages the running servers, deploys code, and makes sure that everyone is happy and has a seat at the table.

The FC is an Azure application in and of itself, running multiple copies of itself for redundancy's sake. It's largely written in managed code. The FC contains the complete state of the fabric internally, which is replicated in real time to all the nodes that are part of the FC. If one of the primary nodes goes offline, the latest state information is available to the remaining nodes, which then elect a new primary node.

The FC manages a state machine for each service deployed, setting a goal state that's based on what the service model for the service requires. Everything the FC does is in an effort to reach this state and then to maintain that state when it's reached. We'll go into the details of what the service model is in the next few pages, but for now, just think of it as a model that defines the needs and expectations that your service has.

The FC is obviously very busy. Let's look at how it manages to seamlessly perform all these tasks.

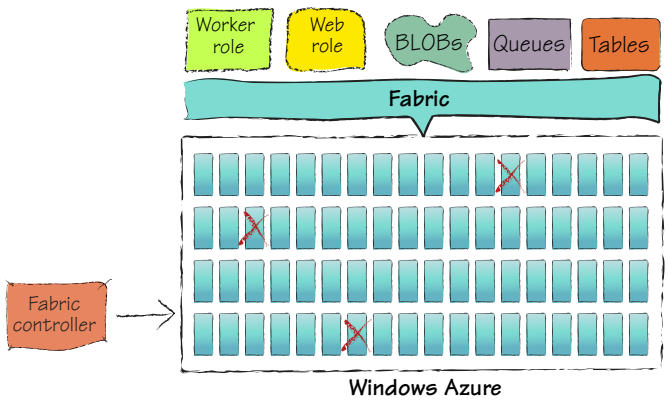

Figure 3.3 The relationship between Azure, the fabric, and the Fabric Controller (FC). The fabric is an abstract model of the massive number of servers in the Azure data center. The FC manages everything. For example, it recovers failed servers and moves your application to a healthy server.

3.3.1 How the FC works: the driver model

The FC follows a driver model, just like a conventional OS. Windows has no idea how to specifically work with your video card. What it does know is how to speak to a video driver, which in turn knows how to work with a specific video card. The FC works with a series of drivers for each type of asset in the fabric. These assets include the machines, as well as the routers, switches, and load balancers.

Although the variability of the environment is low today, over time new types of each asset are likely to be introduced. The goal is to reduce unnecessary diversity, but you'll have business needs that require breadth in the platform. Perhaps you'll get a software load balancer for free, but you'll have to pay a little bit more per month to use a hardware load balancer. A customer might choose a certain option, such as a hardware load balancer, to meet a specific need. The FC would have a different driver for each piece of infrastructure it controls, allowing it to control and communicate with that infrastructure.

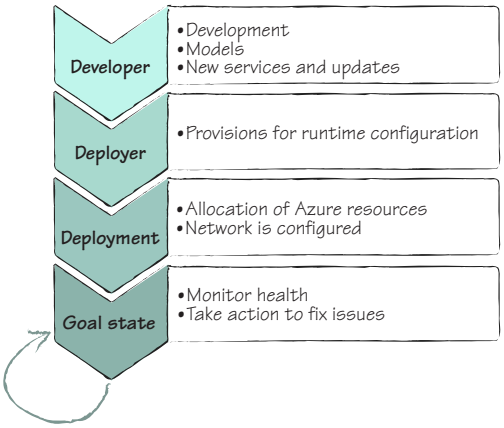

Figure 3.4 How the lifecycle of an Azure service progresses towards a running state. Each role on your team has a different set of responsibilities. From here the FC does what it needs to make sure your servers are always running.

The FC uses these drivers to send commands to each device that help these devices reach the desired running state. The commands might create a new VLAN to a switch or allocate a pool of virtual IP addresses. These commands help the FC move the state of the service towards the goal state. Figure 3.4 shows how a service progresses to the goal state, from the developer writing the code and defining the service model to the FC allocating and managing the resources the service requires.

While the FC is moving all your services toward the running state, it's also allocating resources and managing the health of the nodes in the fabric and of your services.

3.3.2 Resource allocation

One of the key jobs of the FC is to allocate resources to services. It analyzes the service model of the service, including the fault and update domains, and the availability of resources in the fabric. Using a greedy resource allocation algorithm, it finds which nodes can support the needs of each instance in the model. When it has reserved the capacity, the FC updates its data structures in one transaction. After the update, the goal state of each node is changed, and the FC starts moving each node towards its goal state by deploying the proper images and bits, starting up services, and issuing other commands through the driver model to all the resources needed for the change.

3.3.3 Instance management

The FC is also responsible for managing the health of all of the nodes in the fabric, as well as the health of the services that are running. If it detects a fault in a service, it tries to remediate that fault, perhaps by restarting the node or taking it offline and replacing it with a different node in the fabric.

When a new container is added to the data center, the FC performs a series of burn-in tests to ensure that the hardware delivered is working correctly. Part of this process results in the new resource being added into the inventory for the data center, making it available to be allocated by the FC.

If hardware is determined to be faulty, either during installation or during a fault, the hardware is flagged in the inventory as being unusable and is left alone until later. When a container has enough failures, the remaining workloads are moved to different containers and then the whole container is taken offline for repair. After the problems have been fixed, the whole container is retested and returned into service.

3.4 The service model and you

The driving force behind what the FC does is the service model that you define for your service (see figure 3.5). You define the service model indirectly by defining the following things when you're developing a service:

- Some configuration about what the pieces to your service are
- How the pieces communicate
- Expectations you have about the availability of the service

The service model is broken into two pieces of configuration and is deployed with your service. Each piece focuses on a different aspect of the model. In the following sections, you're going to learn about these configuration pieces and how to customize them. We'll also show you how best to manage all the pieces of your configuration.

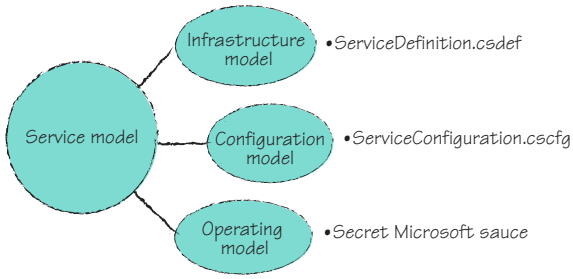

Figure 3.5 The service model consists of several different pieces of information. This model helps Azure run your application correctly.

3.4.1 Defining configuration

Your solution in Visual Studio contains these two pieces of configuration in different files, both of which are found in the Azure Service project in your solution:

- Service definition file (ServiceDefinition.csdef)
- Service configuration file (ServiceConfiguration.cscfg)

The service definition file defines what the roles and their communication endpoints are in your service. This includes public HTTP traffic for a website, or the endpoint

details for a web service. You can also configure your service to use local storage (which is different from Azure storage) and any custom configuration elements of the service configuration file. The service definition can't be changed at runtime; any change requires a new deployment of your service. Your service is restricted to using only the network endpoints and resources that are defined in this model. We're going to look at the service definition file in depth in chapter 4; for now you can think of this piece of the configuration as defining what the infrastructure of your service is, and how the parts fit together.

The service configuration file, which we'll discuss in detail in chapter 5, includes the entire configuration needed for the role instances in your service. Each role has its own dedicated part of the configuration. The contents of the configuration file can be changed at runtime, which removes the need to redeploy your application when some part of the role configuration changes. You can also access the configuration in code, similar to how you might read a web.config file in an ASP.NET application.

3.4.2 Adding a custom configuration element

In many applications, you store connection strings, default settings, and secret passwords (please don't!) in the app.config or web.config file. You'll often do the same with an Azure application. First, you need to declare the format of the new configuration setting in the .csdef file by adding a `ConfigurationSettings` node inside the role you want the configuration to belong to:

```
<ConfigurationSettings>
  <Setting name="BannerText"/>
</ConfigurationSettings>
```

Adding this node defines the schema of the .cscfg file for that role, which strongly types the configuration file itself. If there's an error in the configuration file during a build, you'll receive a compiler warning. This is a great feature because there's nothing worse than deploying code when there's a simple little problem in a configuration file.

Now that you've told Azure the new format of your configuration files, namely, that you want a new setting called `BannerText`, you can add that node to the service configuration file. Add the following XML into the appropriate role node in the .cscfg file:

```
<ConfigurationSettings>
  <Setting name="BannerText" value="KlatuBaradaNikto"/>
</ConfigurationSettings>
```

During runtime, you want to read in this configuration data and use it for some purpose. Remember that all configuration settings are stored as strings and must be cast to the appropriate type as needed. In this case, you want a string to assign to your label control text, so that you can use it as is.

```
txtPassword.Text = RoleEnvironment.GetConfigurationSettingValue("BannerText");
```

Having lines of code like this all over your application can get messy and hard to manage. Sometimes developers consolidate their configuration access code into one class. This class's only job is to be a façade into the configuration system.

3.4.3 Centralizing file-reading code

It's a best practice to move your entire configuration file-reading code from wherever it's sprinkled into a `ConfigurationManager` class of your own design. Many people use the term *service* instead of *manager*, but we think that the term *service* is too overloaded and that *manager* is just as clear. Moving your code centralizes all the code that knows how to read the configuration in one place, making it easier to maintain. More importantly, it removes the complexity of reading the configuration from the relying code, which illustrates the principle of *separation of concerns*. Moving the code to a centralized location also makes it easier to mock out the implementation of the `ConfigurationManager` class for easier testing purposes (see figure 3.6). Over time, when the APIs for accessing configuration change or if the location of your configuration changes, you'll have only one place to go to make the changes you need.

Reading configuration data in this manner might look familiar to you. You've probably done this for your current applications, reading in the settings stored in a web.config or an app.config file. When migrating an existing application to Azure, you might be tempted to keep the configuration settings where they are. Although keeping them in place reduces the amount of change to your code as you migrate it to Azure, it does come at a cost. Unfortunately, the configuration files that are part of your roles are frozen and are read-only at runtime; you can't make changes to them after your package is deployed. If you want to change settings at runtime, you'll need to store those settings in the .cscfg file. Then, when you want to make a change, you only have to upload a new .cscfg file or click Configure on the service management page in the portal.

The FC takes these configuration files and builds a sophisticated service model that it uses to manage your service. At this time, there are about three different core model templates that all other service models inherit from. Over time, Azure will expose more of the service model to the developer, so that you can have more fine-grained control over the platform your service is running on.

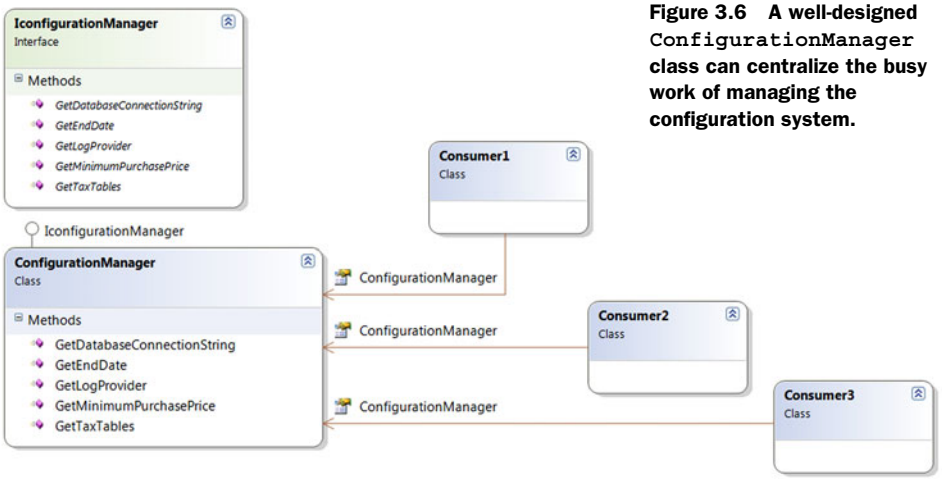

Figure 3.6 A well-designed `ConfigurationManager` class can centralize the busy work of managing the configuration system.

3.4.4 The many sizes of roles

Each role defined in your service model is basically a template for a server you want to be deployed in the fabric. Each role can have a different job and a different configuration. Part of that configuration includes local storage and the number of instances of that role that should be deployed. How these roles connect and work together is part of why the service model exists.

Because each role might have different needs, there are a variety of VM sizes that you can request in your model. Table 3.1 lists each VM size. Each step up in size doubles the resources of the size below it.

Table 3.1 The available sizes of the Azure VMs

VM size	Dedicated CPU cores	Available memory	Local disk space
Small	1	1.7 GB	250 GB
Medium	2	3.5 GB	500 GB
Large	4	7 GB	1,000 GB
Extra large	8	15 GB	2,000 GB

Each size is basically a slice of how big a physical server is, which makes it easy to allocate resources and keeps the numbers round. Because each physical server has eight CPU cores, allocating an extra-large VM to a role is like dedicating a whole physical machine to that instance. You'll have all the CPU, RAM, and disk available on that machine. Which size you want is defined in the ServiceDefinition.csdef file on a role-by-role basis. The default size, if you don't declare one, is small. To change the default size, add the following code, substituting `ExtraLarge` with the size that you want:

```
<WorkerRole name="ImageCompresser" vmsize="ExtraLarge">
```

If you're using Visual Studio 2010, you can define the role configuration by double-clicking the name of your web role in the Roles folder of your Cloud Service project. Choose Properties and click the Configuration tab, as shown in figure 3.7.

The service model is also used to define fault domains and update domains, which we'll look at next.

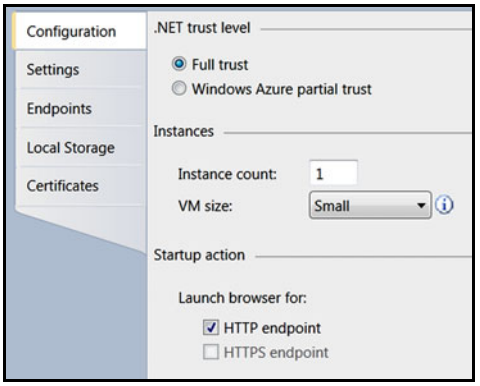

Figure 3.7 Configuring your role doesn't have to be a gruesome XML affair. You can easily do it in Visual Studio 2010 when you view the properties information for the role you want to configure.

3.5 It's not my fault

Fault domains and update domains determine what portions of your service can be offline at the same time, but for different reasons. They're the way that you define your uptime requirements to the FC and how you describe how your service updates will happen when you have new code to deploy.

Let's examine each type of domain in detail. Then we'll present a service model scenario that shows you how fault and update domains help increase fault tolerance in your cloud service.

3.5.1 Fault domains

Fault domains are used to make sure that a set of elements in your service isn't tied to a single point of failure. Fault domains are based more on the physical structure of the data center than on your architecture. Your service should typically have three or more fault domains. If you have only one fault domain, all the parts of your service could potentially be running on one rack, in the same container, connected to the same switch. If there's any failure in that chain, there's a high likelihood of catastrophic failure for your service. If that rack fails, or the switch in use fails, then your service is completely offline. By breaking your service into several fault domains, the FC ensures that those fault domains don't share any dependent infrastructure, which protects your service against single points of failure.

In general, the FC will define three fault domains, meaning that only about a third of them can become unavailable because of a single fault. In a failure scenario, the FC immediately tries to deploy your roles to new nodes in the fabric to make up for the failed nodes. Currently, the Azure SDK and service model don't let you define your own number of fault domains; the default number is thought to be three domains.

3.5.2 Update domains

The second type of domain defined in the service model is the *update domain*. The concept of an update domain is similar to a fault domain. An update domain is the unit of update you've declared for your service. When performing a rolling update, code changes are rolled out across your service one update domain at a time. Cloud services tend to be big and tend to always need to be available. The update domain allows a rolling update to be used to upgrade your service, without having to bring the entire service down. These domains are usually defined to be orthogonal to your fault domains. In this manner, if an update is being pushed out while there's a massive fault, you won't lose all of your resources, just a piece of them.

You can define the number of update domains for your service in your ServiceDefinition.csdef file as part of the `ServiceDefinition` tag at the top of the file.

```
<ServiceDefinition xmlns="http://schemas.microsoft.com/ServiceHosting/
2008/10/ServiceDefinition"
name="HawaiianShirtShop"
upgradeDomainCount="3">
```

If you don't define your own update domain setting, the service model will default to five update domains. Your role instances are assigned to update domains as they're started up, and the FC tries to keep the domains balanced with regard to how many instances are in each domain.

3.5.3 A service model example

If you had a service running on Azure, you might need six role instances to handle the demand on your service, but you should request nine instances instead. You request more than you need because you want a high degree of tolerance in your architecture. As shown in figure 3.8, you would have three fault domains and three update domains defined. If there's a fault, only a third of your nodes are affected. Also, only a third of the nodes will ever be updated at one time, controlling the number of nodes taken out of service for updates, as well as reducing the risk of any update taking down the whole service.

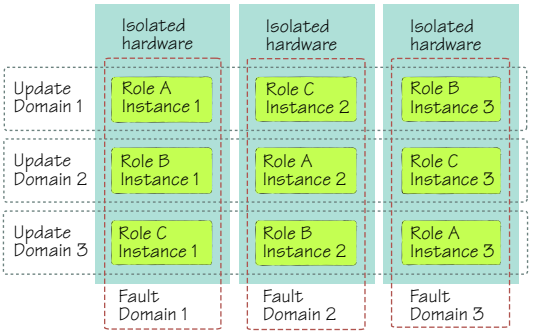

Figure 3.8 Fault and update domains help increase fault tolerance in your cloud service. This figure shows three instances of each of three roles.

In this scenario, a broken switch might take down the first fault domain, but the other two fault domains would not be affected and would keep operating. The FC can manage these fault domains because of the detailed models it has for the Azure data center assets.

The cloud is not about perfect computing, it's about deploying services and managing systems that are fault tolerant. You need to plan for the faults that are inevitable. The magic of cloud computing makes it easy to scale big enough so that a few node failures don't really impact your service.

All this talk about service models and an overlord FC is nice, but at the end of the day, the cloud is built from individual pieces of hardware. There's a lot of hardware, and it all needs to be managed in a hands-off way. There are several approaches to applying updates to a service that's running. You'll see in the next section that you can perform either manual or automated rolling upgrades, or you can perform a full static upgrade (also called a VIP swap).

3.6 Rolling out new code

No matter how great your code is, you'll have to perform an upgrade at some point if for no other reason than to deploy a new feature a user has requested. It's important that you have a plan for updating the application and have a full understanding of the moving parts. There are two major ways to roll out an upgrade: a *static upgrade* or a *rolling upgrade*.

When you perform a static upgrade, you do everything at once and you have to take down your system, at least for a while. You should carefully plan your application architecture to avoid a static upgrade because it impacts the uptime of your service and can be more complicated to roll out. A rolling upgrade keeps your service up and running the whole time. You should always consider performing the upgrade in the staging environment first to make sure the deployment goes well. After a full battery of end-to-end and integration tests are passed, you can proceed with your plans for the production environment.

If the number of endpoints for a role has changed, or if the port numbers have changed, you won't be able to do either a static or a rolling upgrade. You'll be forced to tear down the deployment and redeploy.

3.6.1 Static upgrades

A static upgrade is sometimes referred to as a *forklift upgrade* because you're touching everything all at once. You usually need to do a static upgrade when there's a significant change in the architecture and plumbing of your application. Perhaps there's a whole new architecture of how the services are structured and the database has been completely redesigned. In this case, it can be hard to upgrade just one piece at a time because of interdependencies in the system. This type of upgrade is required if you're changing the service model in any way.

This approach is also called a *VIP swap* because the FC is swapping the virtual IP addresses that are assigned to your resources. When a swap is done, the old staging environment becomes your new production environment and your old production environment becomes your new staging environment (see figure 3.9). This can happen pretty fast, but your service will be down while it's happening and you need to

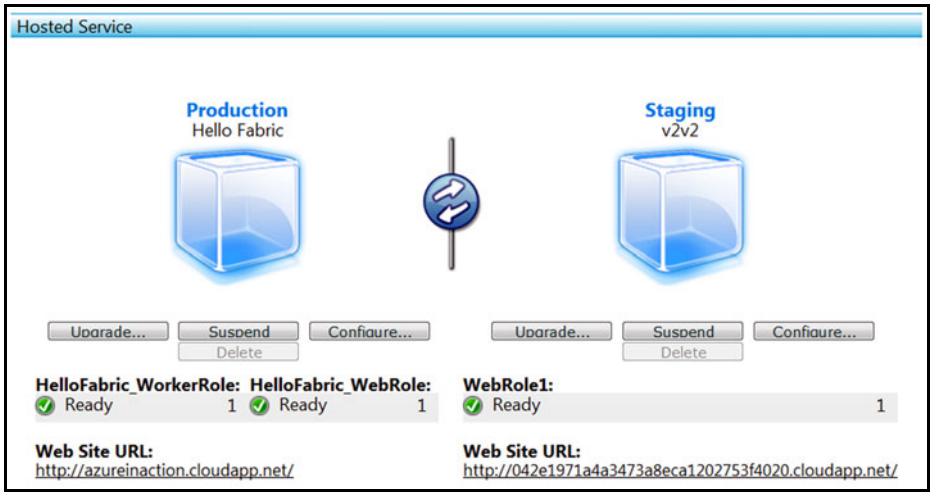

Figure 3.9 Performing a VIP swap, or static upgrade, is as easy as clicking the arrows. If things go horribly awry, you can always swap back to the way things were. It's like rewind for your environment.

plan for that. The one great advantage to this approach is that you can easily swap things back to the way they were if things don't work out.

Your upgrade plan should consider how long the new staging (aka old production) environment should stay around. You might want to keep it for a few days until you know the upgrade has been successful. At that point, you can completely tear down the environment to save resources and money.

To perform a VIP swap, log in to the Azure portal, choose the service that you want to upgrade in the Windows Azure section, and then click the Summary tab. Next, deploy your new application version to the staging environment. After everything is all set up and you're happy with it, click the circular button in the middle. The changeover takes only a few minutes. If the new version isn't working as expected, you can easily click the button again and swap the two environments back where they came from. Voila! The old version is back online.

You can also use the service management API to perform the swap operation. This is one reason why you want to make sure that you've named your deployments clearly, at least more clearly than we did in this example.

VIP swaps are nice, but some customers need more flexibility in the way they perform their rollouts. For them, there's the rolling upgrade.

3.6.2 Rolling upgrades

If your roles are carrying state and you don't want to lose that state as you completely change to a new set of servers, then rolling upgrades are for you. Or maybe you want to upgrade the instances of a specific role instead of all of the roles. For example, you might want to deploy an updated version of the website, without impacting the processing of the shopping carts that's being performed by the backend worker roles. Remember that when doing a rolling upgrade, you can't change the service model of the service that you're upgrading. If you've changed the structure of the service configuration, the number of endpoints, or the number of roles, you'll have to do a VIP swap instead.

There are two types of rolling upgrades: the automatic and the manual. When you perform an automatic rolling upgrade, the FC drains the traffic to the set of instances that's in the first update domain (they're numbered, starting with 0) by removing them from the load balancer's configuration. After the traffic is drained, the instances are stopped, the new code is deployed, and then the instances are restarted. After they're back up and running, they're added back into the load balancer's list of machines to route traffic to. At this point, the FC moves on to the next update domain in the list. It'll proceed in this fashion until all the update domains have been serviced. Each domain should take only a few minutes.

If your situation requires that you control how the progression moves from one domain to the next, you can choose to do a manual rolling upgrade. When you choose this option, the FC stops after updating a domain and waits for your permission to move on to the next one. This gives you a chance to check the status of the machines and the environment before moving forward with the rollout.

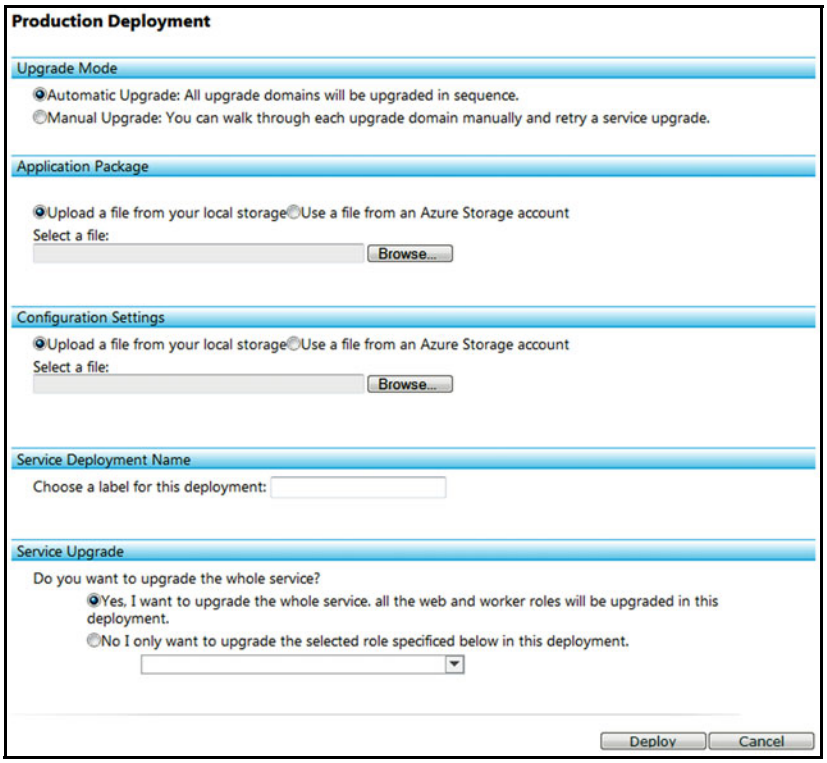

Figure 3.10 Performing a rolling upgrade is easy. Click Upgrade on the Summary page for the service to see this page and choose your options. You can upgrade all of the roles or just one role during an upgrade.

To perform a rolling upgrade, log in to the Azure portal, choose the service that you want to upgrade in the Windows Azure section, and then click the Summary tab. Click the Upgrade button for the deployment you want to upgrade. You're presented with some options, as shown in figure 3.10.

You can choose to perform an automatic or a manual upgrade. You can upgrade all the roles in the package or just one of them. As in a normal deployment, you also need to provide a service package, configuration, and a deployment name.

If you choose to upgrade a single role, then only the instances for that role in each domain are taken offline for upgrading. The other role instances are left untouched.

You can also perform a rolling upgrade by using the service management API. When you use the management API, you have to store the package in BLOB storage before starting the process. As with a VIP swap, you need to post a command to a specific URL (all these commands are covered in detail in chapter 18). Customize the URL to match the settings for the deployment you want to upgrade:

```
https://management.core.windows.net/<subscription-id>/services/
     hostedservices/<service-name>
/deployments/<deployment-name>/?comp=upgrade
```

The body of the command needs to contain the elements shown in the following code. You need to change the code to supply the parameters that match your situation. The following sample performs a fully automatic upgrade on all the roles.

```
<?xml version="1.0" encoding="utf-8"?>
<UpgradeDeployment xmlns="http://schemas.microsoft.com/windowsazure">
  <Mode>auto</Mode>
  <PackageUrl>http://azureinaction.blob.core.windows.net/
deployment_container/new_code.cspkg </PackageUrl>
  <Configuration>***the contents of the config file***</Configuration>
  <Label>v3.2</Label>
</UpgradeDeployment>
```

Performing a manual rolling update with the service management API is a little trickier, and requires several calls to the `WalkUpgradeDomain` method. The upgrades are performed in an asynchronous manner; the first command starts the process. As the upgrade is being performed, you can check on the status by using `Get Operation Status` with the operation ID that was supplied to you when you started the operation.

We've covered how to upgrade running instances and talked about what the fabric is. Now we'll go one level deeper and explore the underlying environment.

3.7 The bare metal

No one outside of the Azure team truly knows the nature of the underlying servers and other hardware, and that's OK because it's all abstracted away by the cloud OS. But you can still look at how your instances are provisioned and how automation is used to do this without hiring the entire population of southern Maine to manage it.

Each instance is really a VM running Windows Server 2008 Enterprise Edition x64 bit, on top of *Hyper-V.* Hyper-V is Microsoft's enterprise virtualization solution, and it's available to anyone. Hyper-V is based on a hypervisor, which manages and controls the virtual servers running on the physical server. One of the virtual servers is chosen to be the host OS. The host OS is a virtual server as well, but it has the additional responsibilities of managing the hypervisor and the underlying hardware.

Hyper-V has two features that help in maximizing the performance of the virtual servers, while reducing the overall cost of running those servers. One of these features is core-and-socket parking; the other is the reduced footprint of Hyper-V itself. Core-and-socket parking needs to be supported by the physical CPU.

Let's drill way down into the workings of Hyper-V, how the virtual servers connect to it, and the processes of booting up these servers and getting your instances up and running.

3.7.1 Free parking

The first feature of Hyper-V is core-and-socket parking. Hyper-V can monitor the use of each core and CPU (which is in a socket on the motherboard) as a whole. Hyper-V moves the processes around on the cores to consolidate the work to as few cores as possible. Any cores not needed at that time are put into a low energy state. They can come back online quickly if needed but consume much less power while they wait.

Hyper-V can do this at the socket level as well. If it notices that each CPU socket is being used at only 10 percent of capacity, for example, it can condense the workload to one socket and park the unused sockets, placing them in a low energy state. This helps data centers use less power and require less cooling. In Azure, you have exclusive access to your assigned CPU core. Hyper-V won't condense your core with someone else's. It will, however, turn off cores that aren't in use.

3.7.2 A special blend of spices

The version of Hyper-V used by Azure is a special version that the team created by removing anything they didn't need. Their goal was to reduce the footprint of Hyper-V as much as possible to make it faster and easier to manage. Because they knew exactly the type of hardware and guest operating systems that will run on it, they could rip out a lot of code. For example, they removed support for 32-bit hosts and guest machines, support needed for other types of operating systems, and support for hardware they weren't supporting at all.

Not stopping there, they further tuned the hypervisor scheduler for better performance while working with cloud data-center workloads. They wanted the scheduler to be more predictable in its use of resources and fairer to the different workloads that were running, because each would be running at the same priority level. They also enhanced Hyper-V to support a heavier I/O load on the VM bus.

3.7.3 Creating instances on the fly

When a new server is ready to be used, it's booted. At this point, it's a naked server with a bare hard drive. You can see the steps involved in starting the server, adding an instance to your service, and adding an additional server in figure 3.11.

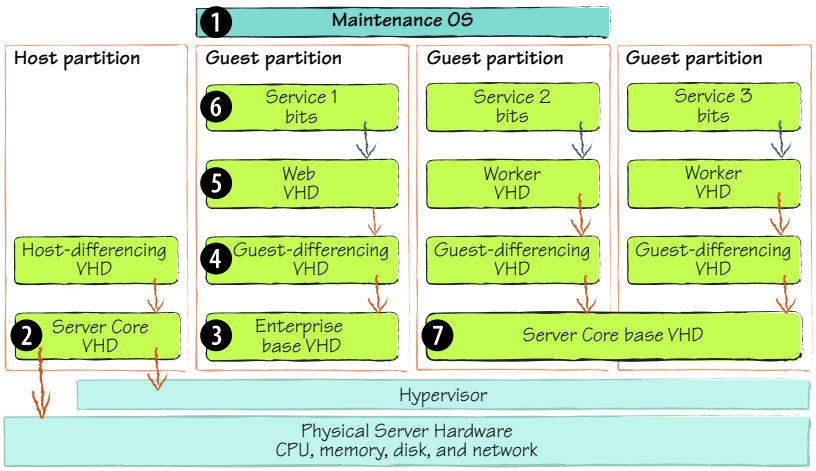

Figure 3.11 The structure of a physical server and virtual instance servers in Azure. This figure illustrates the process involved in starting the server (1 and 2). It also shows the process of starting an instance and adding it to your service (3 through 6), and adding another virtual server (7). All these steps are coordinated by the FC and take only a few minutes.

During boot up, the server locates a maintenance OS on the network, using standard Preboot Execution Environment (PXE) protocols. (PXE is a process for booting to an operating system image that can be found on the network.) The server downloads the image and boots to it (1 in figure 3.11).

The maintenance OS

The maintenance OS is based on Windows Preinstallation Environment (Windows PE). It's a thin OS that's used by many IT organizations for low-level troubleshooting and maintenance. The tools and protocols for Windows PE are available on any Windows server and are used by a lot of companies to easily distribute machine images and automate deployment.

The maintenance OS connects with the FC and acts as an agent for the FC to execute any local commands needed to prepare the disk and machine. The agent prepares the local disk and streams down a Windows Server 2008 Server Core image to the local disk 2. This image is a virtual hard drive (VHD) and is a common file format used to store the contents of hard drives for VMs. (VHDs are large files representing the complete or partial hard drive for a VM.) The machine is then reconfigured to boot from this core VHD. This image becomes the host OS that manages the machine and interacts with the hypervisor. The host OS is Windows Server 2008 Core because almost all but the most necessary modules have been removed from the operating system. You might be running this in your own data center.

The Azure team worked with the Windows Server team to develop the technology needed to boot a machine natively from a VHD that's stored on the local hard drive. The Windows 7 team liked the feature so much that they added it to their product as well. Being able to boot from a VHD is a key component of the Azure automation.

After the machine has rebooted using the host OS image, the maintenance OS is removed and the FC can start allocating resources from the machine to services that need to be deployed. A base OS image is selected from the prepared image library that'll meet the needs of the service that's being deployed 3. This image (a VHD file) is streamed down to the physical disk. The core OS VHDs are marked as read-only, allowing multiple service instances to share a single image. A differencing VHD is stacked on top of the read-only base OS VHD to allow for changes specific to that virtual server 4. Different services can have different base OS images, based on the service model applied to that service.

On top of the base OS image and attached to it is an application VHD that contains additional requirements for your service 5. The bits for your service are downloaded to the application VHD 6, and then the stack is booted. As it starts, the stack reports its health to the FC. The FC then enrolls the stack into the service group, configuring the VLAN assigned to your service and updating the load balancer, IP allocation, and DNS

configuration. When this process is completed, the new node is ready to service requests to your application.

Much of the image deployment can be completed before the node is needed, cutting down on the time it can take to start a new instance and add it to your service.

Each server can contain several VMs. This allows for the optimal use of computing resources and the flexibility to move instances around as needed. As a second or third virtual server is added, it might use the base OS VHD that has already been downloaded ❼ or it can download a different base OS VHD based on its needs. This second machine then follows the same process of downloading the application VHD, booting up, and enrolling into the cloud.

All these steps are coordinated by the FC and are usually accomplished in a few minutes.

3.7.4 Image is everything

If the key to the automation of Azure is Hyper-V, then the base VM images and their management are the cornerstone. Images are centrally created, also in an automated fashion, and stored in a library, ready to be deployed by the FC as needed.

A variety of images are managed, allowing for the smallest footprint each role might need. If a role doesn't need IIS, then there's an image that doesn't have IIS installed. This is an effort to shrink the size and runtime footprint of the image, but also to reduce any possible attack surfaces or patching opportunities.

All images are deployed using an Xcopy deployment model. This model keeps deployment simple. If the FC relied on complex scripts and tools, then it would never truly know what the state of each server would be and it would take a lot longer to deploy an instance. Again, diversity is the devil in this environment.

This same approach is used when deploying patches. When the OS needs to be patched, Microsoft doesn't download and execute the patch locally as you might on your workstation at home. Doing so would lead to too much risk, having irregular results on some of the machines. Instead, the patch is applied to an image and the new image is stored in the library. The FC then plans a rollout of the upgrade, taking into account the layout of the cloud and the update domains defined by the various service models that are being managed.

The updated image is copied in the background to all of the servers used by the service. After the files have been staged to the local disk, which can take some time, each update domain group is restarted in turn. In this way, the FC knows exactly what's on the server. The new image is merely wired up to the existing service bits that have already been copied locally. The old image is kept locally for a period of time as an escape hatch in case something goes wrong with the new image. If that happens, the server is reconfigured to use the old image and rebooted, according to the update domains in the service model. This process dramatically reduces the service window of the servers, increasing uptime and reducing the cost of maintenance on the cloud.

We've covered how images are used to manage the environment. Now we're going to explain what you can see when you look inside a running role instance.

3.8 The innards of the web role VM

Your first experience with roles in Azure is likely to be with the web role. To help you develop your web applications more effectively, it's worth looking in more detail at the VM that your web role is hosted in. In this section, we'll look at the following items:

- The details of the VM
- The hosting process of your web role (`WaWebHost`)
- The `RDAgent` process

3.8.1 Exploring the VM details

You can use the power of native code execution to see some of the juicy details about the VM that your web role runs on. Figure 3.12 shows an ASP.NET web page that shows some of the internal details, including the machine name, domain name, and the user name that the code is running under.

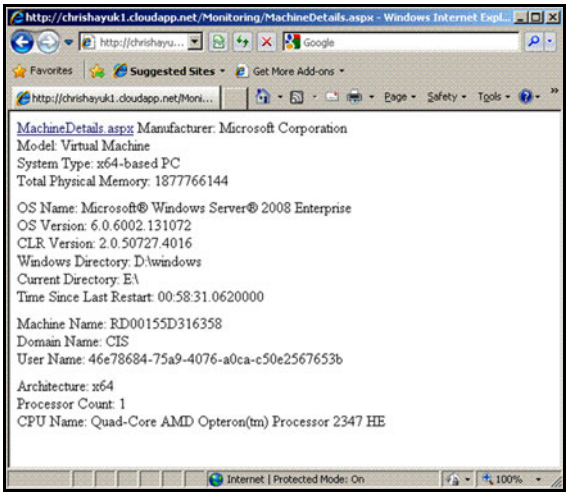

Figure 3.12 Using native code, you can see some of the machine details of a web role in Windows Azure. In this example, Microsoft is using Windows Server 2008 Enterprise x64. Notice that the user name that the process is running as is a GUID.

If you want, you can easily generate the web page shown in figure 3.12 by creating a simple ASPX page with some labels that represent the text, as follows:

```
<div>
   ProcessorCount:
            <asp:Label ID="lblProcessorCount" runat="server" />
 </div>
```

Finally, you can display the internal details of the VM using the code-behind in the following listing.

Listing 3.1 Using code behind to display machine details

```
using System.Management;

public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

            // Initialize
            var computer = new Microsoft.VisualBasic.Devices.Computer();
            lblMachineName.Text = computer.Name;
```

Class fetches information about server

```
    // OS Details
    lblOSName.Text = computer.Info.OSFullName;
    lblOSVersion.Text = computer.Info.OSVersion;
    lblMachineName.Text = computer.Name;

    // Computer System Details
    lblProcessorCount.Text =
    ➥ System.Environment.ProcessorCount.ToString();
    lblCLRVersion.Text = System.Environment.Version.ToString();
    lblCurrentDirectory.Text = GetCurrentDirectory();
    lblTimeSinceLastRestart.Text = GetTimeSinceLastRestart();
    lblDomainName.Text = System.Environment.UserDomainName;
    lblUserName.Text = System.Environment.UserName;
    lblCPUName.Text = GetCPUName();
    lblArchitecture.Text = GetArchitecture();
}

private string GetCurrentDirectory()
{
    try
    {
        return System.Environment.CurrentDirectory;
    }
    catch
    {
        return "unavailable";
    }
}

private string GetTimeSinceLastRestart()
{
    try
    {
        TimeSpan time = new TimeSpan(0, 0, 0, 0,
        ➥ System.Environment.TickCount);
        return time.ToString();
    }
    catch
    {
        return "unavailable";
    }
}

private string GetCPUName()
{
    try
    {
        using (ManagementObject Mo = new
        ➥ ManagementObject("Win32_Processor.DeviceID='CPU0'"))
        {
            return (string)(Mo["Name"]);
        }
    }
    catch
    {
        return "unavailable";
```

Gets length of time server has been running

Gets user name service is running as

Gets domain name server is running on

```
            }
        }

        private string GetArchitecture()
        {
            try
            {
                using (ManagementObject Mo = new
                ➥ ManagementObject("Win32_Processor.DeviceID='CPU0'"))
                {
                    ushort result = (ushort)(Mo["Architecture"]);
                    switch (result)
                    {
                        case 0:
                            return "x86";
                        case 9:
                            return "x64";
                        default:
                            return "other";
                    }
                }
            }
            catch
            {
                return "unavailable";
            }

        }
    }
```

You can now, of course, deploy your web page to Windows Azure and see the inner details of your web role, which were shown in figure 3.12. These machine details provide you with some interesting facts:

- Web roles run on Windows 2008 Enterprise Edition x64
- They run quad core AMD processors and one core is assigned
- The domain name of the web role is CIS
- This VM has been running for an hour
- The Windows directory lives on the D:\ drive
- The web application lives on the E:\ drive

This is just the beginning; feel free to experiment and discover whatever information you need to satisfy your curiosity about the internals of Windows Azure by using calls similar to those shown in listing 3.1.

3.8.2 The process list

Now that we're rummaging around the VM, it might be worth having a look at what processes are actually running on the VM. To do that, you'll build an ASP.NET web page that'll return all the processes in a pretty little grid, as shown in figure 3.13.

To generate the list shown in figure 3.13, create a new web page in your web role with a `GridView` component called `processGridView`:

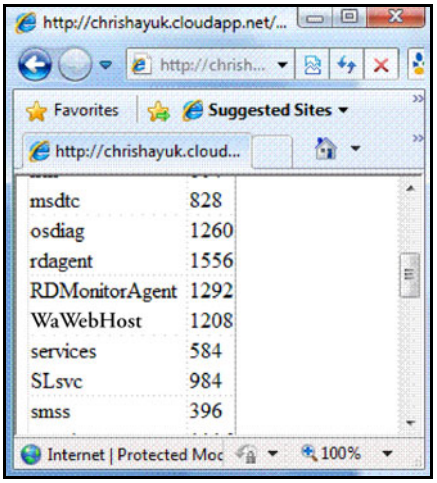

Figure 3.13 The process list of a Windows Azure VM. The RDAgent process is related to Red Dog, which was the code name for Azure while it was being developed.

```
<asp:GridView ID="processGridView" runat="server"/>
```

Next, add a using `System.Diagnostics` statement at the top of the code-behind and then add the following code to the `Page_Load` event:

```
var processes = Process.GetProcesses();

processGridView.DataSource = from process in processes
                             orderby process.ProcessName
                             select new
                             {
                                Name = process.ProcessName,
                                Id = process.Id.ToString() };
                             }
processGridView.DataBind();
```

Gets list of running processes

Uses LINQ query to streamline data returned

Binds query result to grid for screen output

This code will list all the processes on a server and bind the returned list to a `GridView` on a web page, as displayed in figure 3.13. If you look at the process list displayed in figure 3.13, you'll see the two Windows Azure–specific services that we're interested in: `WaWebHost` and `RDAgent`.

We'll now spend the next couple of subtopics looking at these processes in more detail.

3.8.3 The hosting process of your website (WaWebHost)

If you were to look at the process list for a live web role (shown in figure 3.13), or if you were to fire up your web application in Windows Azure and click the Process tab, you would notice that the typical IIS worker process (w3wp.exe) isn't present when your web server is running.

You would also notice that if you stop your IIS server by issuing `IISReset - stop`, your web server continues to run. You know from installing the Windows Azure SDK

that web roles are run under IIS 7.0. So, why can't you see your roles in IIS, or restart the server using `IISReset?`

HOSTABLE WEB CORE

Although Windows Azure uses IIS 7.0, it makes use of a new feature, called *hostable web core,* which allows you to host the IIS runtime in-process. In the case of Windows Azure, the `WaWebHost` process hosts the IIS 7.0 runtime. If you were to look at the process list on the live server or on the development fabric, you would see that as you interact with the web server, the utilization of this process changes.

WHY IS AZURE RUN IN-PROCESS RATHER THAN USING PLAIN OLD IIS?

The implementation of the web role is quite different from that of a normal web server. Rather than using a system administrator to manage the running of the web servers, the data center overlord—the FC—performs that task. The FC needs the ability to interact and report on the web roles in a consistent manner. Instead of attempting to use the Windows Management Instrumentation (WMI) routines of IIS, the Windows Azure team opted for a custom Windows Communication Foundation (WCF) approach.

This custom in-process approach also allows your application instances to interact with the `WaWebHost` processing using a custom API via the `RoleEnvironment` class. You can read more about the `RoleEnvironment` class in chapter 4.

3.8.4 The health of your web role (RDAgent)

The `RDAgent` process collects the health status of the role and the following management information on the VM:

- Server time
- Machine name
- Disk capacity
- OS version
- Memory
- Performance statistics (CPU usage, disk usage)

The role instance and the `RDAgent` process use named pipes to communicate with each other. If the instance needs to notify the FC of its current state of health, notification is communicated from the web role to the `RDAgent` process using the named pipe.

All the information collected by the `RDAgent` process is ultimately made available to the FC; it determines how to best run the data center. The FC uses the `RDAgent` process as a proxy between itself, the VM, and the instance. If the FC decides to shut down an instance, it instructs the `RDAgent` process to perform this task.

3.9 Summary

Hopefully, you've learned a little bit about how Azure is architected and how Microsoft runs the cloud OS. You also know how data centers have changed over the generations

of their development. Microsoft has spent billions of dollars and millions of work hours building these data centers and the OS that runs them.

Windows Azure truly is an operating system for the cloud, abstracting away the details of the massive data centers, servers, networks, and other gear so you can simply focus on your application. The FC controls what's happening in the cloud and acts as the kernel in the operating system. With the power of the FC and the massive data centers, you can define the structure of your system and dynamically scale it up or down as needed. The infrastructure makes it easy to do rolling upgrades across your infrastructure, leading to minimal downtime of your service.

The service model that you define consists of the service definition and service configuration files and describes, to the FC, how your application should be deployed and managed. This model is the magic behind the data center automation. New configuration settings are held in the ServiceDefinition.csdef and ServiceConfiguration.cscfg files. Centralizing all your configuration file-reading code into one neat, handy `ConfigurationManager` class is a real time saver.

Fault and update domains describe how the group of servers running your application should be separated to protect against failures and outages. Fault domains ensure that your service is not tied to a single point of failure, which could be catastrophic to your service. An update domain provides the ability to perform a rolling upgrade, keeping you from having to take down your whole service to do an upgrade.

When you need to upgrade your application, you can perform either a static upgrade or a rolling upgrade, which you can do via the Azure portal. All you do is choose a few options and click a button, or you can use the service management API.

The automated nature of Azure is thanks to Hyper-V, Microsoft's enterprise virtualization solution. Hyper-V consolidates work to as few cores as possible by monitoring the use of each core and CPU, all while maintaining a small footprint.

In the next few chapters, we'll work much more closely with the service runtime. We'll look at how you know when you're running in the fabric, and the configuration magic of the service model.

MICROSOFT .NET/CLOUD

Azure IN ACTION

Chris Hay • Brian H. Prince

Microsoft Azure is a cloud service with good scalability, pay-as-you-go service, and a low start-up cost. Based on Windows, it includes an operating system, developer services, and a familiar data model.

Azure in Action is a fast-paced tutorial that introduces cloud development and the Azure platform. The book starts with the logical and physical architecture of an Azure app, and quickly moves to the core storage services—BLOB storage, tables, and queues. Then, it explores designing and scaling frontend and backend services that run in the cloud. Through clear, crisp examples, you'll discover all facets of Azure, including the development fabric, web roles, worker roles, BLOBs, table storage, queues, and more.

This book requires basic C# skills. No prior exposure to cloud development or Azure is needed.

What's Inside

- Data storage and manipulation
- Using message queues
- Deployment and management
- Azure's data model

A Microsoft MVP specializing in high-transaction databases, **Chris Hay** is a popular speaker and founder of the Cambridge, UK, .NET usergroup. **Brian H. Prince** is a Microsoft Architect Evangelist who helps customers adopt the cloud.

For online access to the authors and a free ebook for owners of this book, go to manning.com/AzureinAction

"Easy to read, easy to recommend."
—Eric Nelson, Microsoft UK

"I doubt even the Azure team knows all of this."
—Mark Monster, Rubicon

"An educational ride at an amusement park—great information and lots of humor."
—Michael Wood
Strategic Data Systems

"Highly recommended, like all Manning books."
—James Hatheway
i365, A Seagate Company

"This book will get you in the cloud... and beyond."
—Christian Siegers, Cap Gemini

ISBN 13: 978-1-935182-48-1
ISBN 10: 1-935182-48-X
54499
9 781935 182481

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]