The Journey of Testing with Stubs and Proxies in AWS

Lucy Chang

lucy_chang@intuit.com

Abstract

Intuit, a leader in small business and accountants software, is a strong AWS(Amazon Web Services) partner and has migrated several services to the AWS platform. We are also implementing a service-oriented architecture to provide better user experiences. With a service-oriented architecture, it is common that the service that one team works on depends on other services, which is referred to as depended-on component (DOC). When the DOC is not ready for integration, we can utilize stubs to simulate the interaction and continue the work. Stubs are canned responses used to replace the real DOC for test-specific purposes. We can create different permutations of test data stubs to increase code coverage. Combining this with the fault injection for resiliency testing helps to ensure the availability and robustness of our services. In addition to stubbing, we also need a proxy to forward the requests to the real DOC for end to end integration testing.

We at the Intuit QuickBooks Online team have researched different options for setting up stubbing and proxies suitable for the unique AWS operating environment. It needs to be easy to learn and easy to adopt. We implemented Wiremock server in AWS for the team and were able to increase our code coverage significantly and do resiliency automation testing which was very difficult to automate before.

This paper describes the following.

1. A basic introduction of architecture design in AWS

2. The Wiremock tool and why it was chosen

3. How to automate setting up Wiremock in AWS

Biography

Lucy Chang is a Senior Software Engineer in Quality at Intuit, currently working in the QuickBooks Online team in Mountain View, California. She has extensive experiences in web services automation testing, performance testing and mobile automation testing.

Lucy has an M.S. in Computer Information Technology from the University of Pennsylvania.

1. Introduction

As the industry embraces the service-oriented architecture, it is common practice that engineers work in a team that focuses on developing a single web service. Most likely this service will depend on other services, which are developed by other teams. Since the depended-on component (DOC) belongs to different teams, it is naturally hard to align the timeline for all the teams to complete all the work by the time engineers need to integrate [1]. Therefore, engineers are forced to wait until the DOCs are ready in order to start the integration work. This affects the release schedule and does not put the engineers' time to the best use when they are blocked and waiting for others to catch up.

What if there was a way that engineers could continue the integration work without the DOC being ready? Using Wiremock to stub the DOC to simulate its designed behavior does exactly that [2]. The application server can communicate to the Wiremock server and the Wiremock server will return stubbed responses as if it is the DOC. This is seamless to the application server and unblocks the engineers for the integration work.

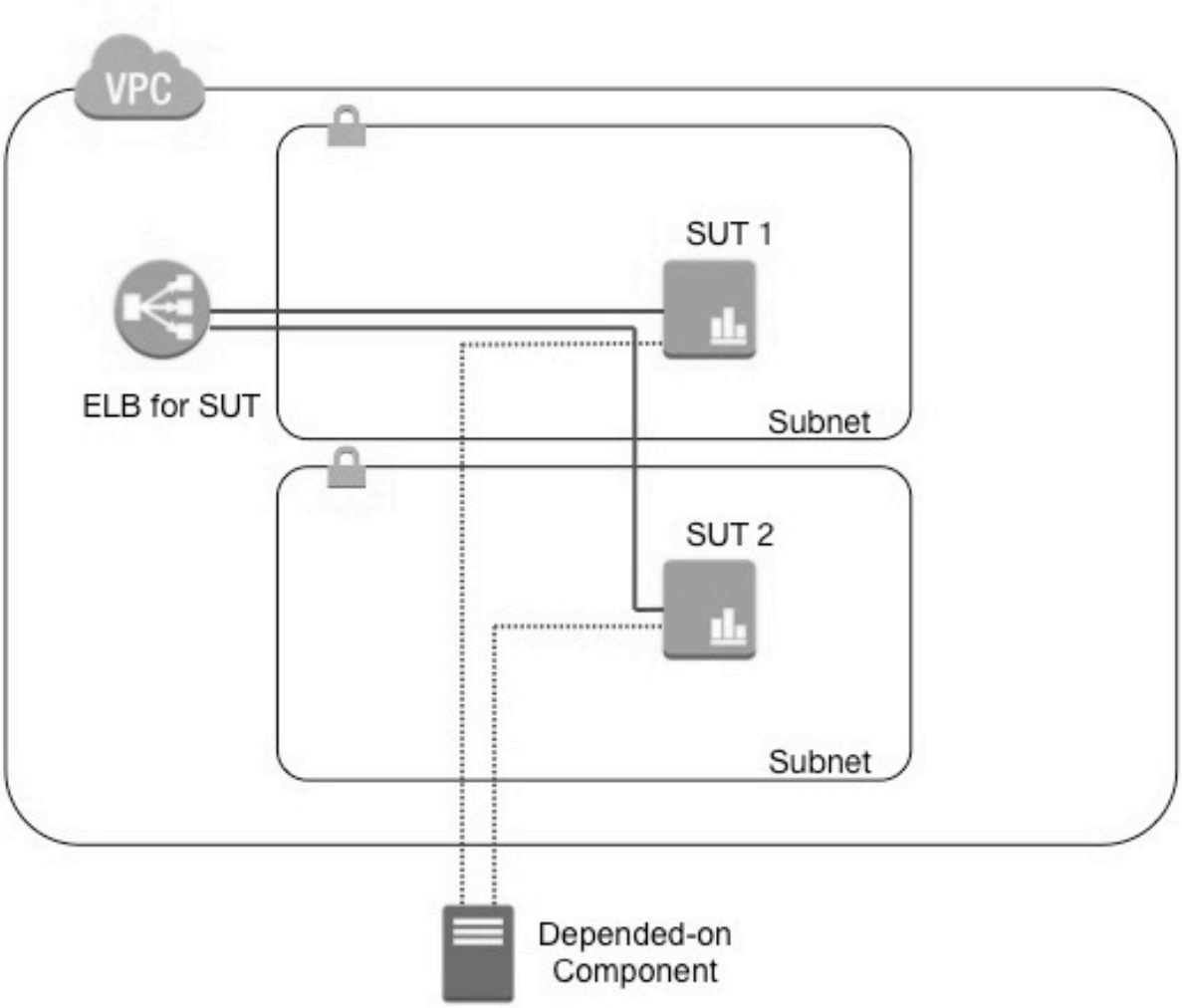
To software quality professionals, this allows for integration and end-to-end testing without relying on the DOC. With stubs, we can test different permutations of responses or negative test cases. Moreover, we can even do fault injection [3] for resiliency testing [4] using stubs in the automation. This opens a new world to the automation code coverage and allows for automation that we were not able to do before. In addition, we still need to do end to end testing with the real DOC. The Wiremock server can act as a proxy to forward the request to the DOC and relay back to the system under test (SUT). We will cover more about how it works in Wiremock section.

Another industry trend is to move the servers to cloud computing services for scalability, availability and the pay-as-you-go pricing model. One of the cloud computing services leader is Amazon Web Services (AWS). Combining these two trends, opportunities arise for testing using stubs and proxies in AWS.

2. AWS

Traditionally, it takes several months to provision new IT hardware before it actually gets into the hands of engineers. With AWS cloud computing services, users now can provision IT resources at any time. Only a few clicks in the AWS console are required to spin up load balancers, Elastic Compute Cloud (EC2) instances, or databases within few minutes. EC2 is a web service that allows users to launch the EC2 instances in the AWS web console. Users can select different capacities, different operating systems, and set up network permission and security access for each EC2 instance. Behind the scenes, an EC2 instance could be a virtual machine or a physical box. This is a huge advantage to the engineers who now have full control of the operating environment. Moreover, using cloud computing services gives the engineers the ability to automate provisioning of IT resources. We will cover more about the automation part in later sections.

Below is a simplified architecture for a system under test (SUT) in AWS. The architecture setup in AWS is built to have high availability, which directly affects the way we set up our Wiremock server. The impact will be discussed later in the paper.



In AWS, we created a Virtual Private Cloud (VPC) dedicated to our AWS account [5]. In our VPC, we can provision AWS resources according to our needs. In this example, we created two subnets located in two different availability zones. Availability zones are distinct physical locations so that when one availability zone is down due to unforeseeable events, for example an earthquake, the AWS resources in other availability zones will not be affected. This significantly increases the availability for our system under test (SUT). Inside each subnet, we launched one EC2 instance and deployed our application on it, which is our SUT. Our SUT talks to the depended-on component (DOC) that is, in this case, outside of our AWS VPC. To distribute the load to the two instances of the SUTs, we set up an Elastic Load Balancing (ELB), which resides in the VPC. The load balancer is responsible for taking the incoming traffic and distributing the requests to the SUTs. All the traffic has to go through the ELB before it is forwarded to the SUT instances.

3. Wiremock

3.1 What is Wiremock

Wiremock is a library for stubbing and proxying web services. It creates an actual HTTP server that integration tests or an application server can connect to as if it is a real web service [6]. After starting up the Wiremock server, we configured the SUT's endpoint URL to point to the Wiremock server instead of the DOC. Then we called the Wiremock API to set up the stub response with a respective request filter. A

request filter can be a specific endpoint, a specific header or a combination of both. If the request matches the request filter, then the Wiremock server will return a stubbed response as if it is from the DOC when the SUT hits the Wiremock server. For the requests that do not match the request filters, the Wiremock server proxies the request to the DOC and relays back. It is seamless to the end users calling the SUT.

3.2 Why Wiremock

3.2.1 The Requirements for Our Team

We were looking for a tool that is able to stub web services requests dynamically from the automation tests. Besides that, it was a plus to allow users to stub without modifying the system under test code except for configuration changes. This empowered the software quality engineers to stub without spending too much time to figure out or set up SUT code. Last but not least, it needed to be easy to learn and straightforward to use.

3.2.2 Why Wiremock was chosen

There are many options for stubbing and proxy tools. Wiremock met all our criteria. Below are the reasons why Wiremock stood out from the others.

1. It requires minimum changes to the SUT code. A simple endpoint URL configuration change pointing to the Wiremock server suffices.

2. It does not require users to dive into the underlying logic for the depended-on component or read the codebase for a system under test. All you need is a sample response and request filter, so that the Wiremock server can return the stub response according to the request filter.

3. It is very easy to set up. After uploading the Wiremock jar to the EC2 instance, a simple one-line command calling the jar will start up the Wiremock server.

4. It allows dynamic stubbing. Users can call the Wiremock API to set the stubs on the fly, which allows users to stub different responses for different test cases in the automation.

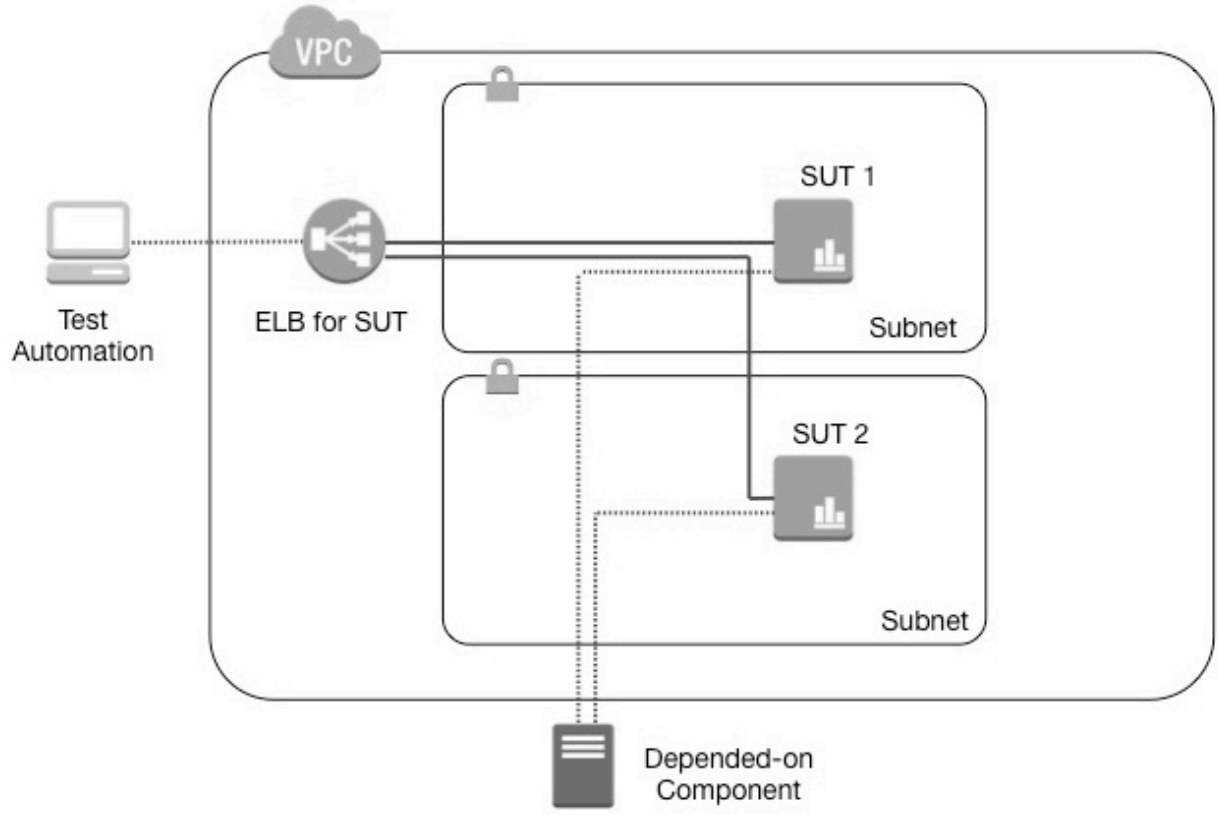
5. The Wiremock API supports fault injection. It empowers us to do resiliency testing for automation that was very difficult to simulate before.

4. Setting Up Wiremock in AWS

4.1 The Architecture in AWS

4.1.1 The Architecture and the Flow Before Setting Up Wiremock

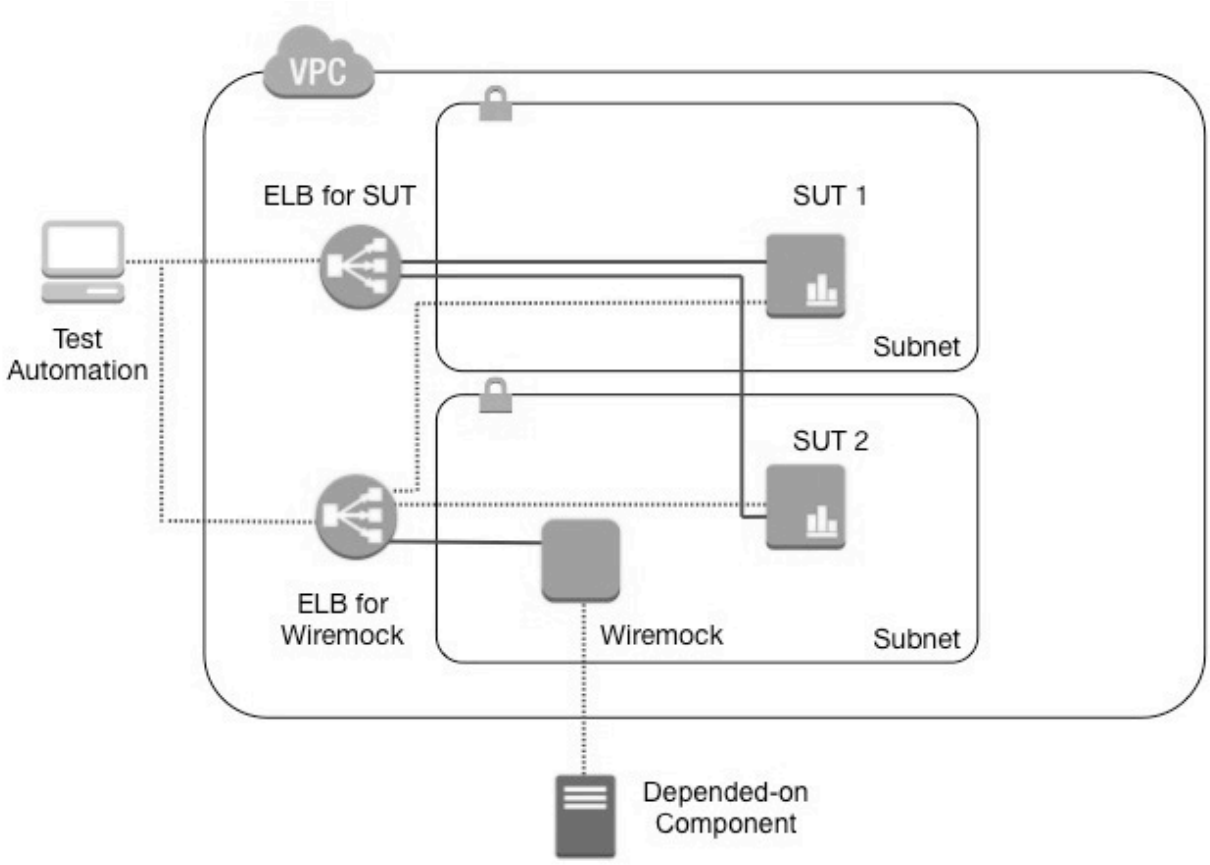
The Test Automation icon in the diagram represents the integration tests, usually written by software quality professionals. SUT 1 and 2 are the systems under test. Before using Wiremock, the automation tests called the ELB for API testing. The ELB forwarded the requests to one of the SUT instances that will in turn call the depended-on component.



4.1.2 The Architecture and the Flow After Setting Up Wiremock

We provisioned an EC2 instance in one of the subnets and deployed the Wiremock jar into the instance. We intentionally chose not to deploy the Wiremock jar into the system under test EC2 instances even though it is more cost efficient and saves all the set up effort for the new additions of the AWS resources. Imagine for a moment that we deploy the Wiremock into SUT 1. The automation calls the Wiremock API to set up stubs. After the ELB receives the Wiremock API request, it can forward the API request to SUT 1 or SUT 2. Let us say it happens to proxy the requests to SUT 1 and sets up the stubs successfully in SUT 1. When the automation hits the ELB again to run the test, the ELB forwards the requests to SUT 2. Since we set up the stubbed response for SUT 1, not SUT 2, when the automation tests hit the SUT 2, the tests will not get the stubs and the tests will fail. Due to the nature of the distributed system of AWS

has, there is no way to set up stubs and get a consistent stubbed response back if we deploy Wiremock to the SUT instance. Thus, we spun up a dedicated EC2 instance for the Wiremock server.



In addition to the dedicated EC2 instance, we provisioned a dedicated ELB for the Wiremock EC2 instance. In AWS, you cannot specify which instance to forward to for ELB. If we reuse the ELB, then the requests might be forwarded to SUT instances instead of the Wiremock instances. There are other benefits for setting up its own ELB. First, we can set up a domain name server (DNS) name for the Wiremock ELB. Some companies have the policy to delete the AWS resources periodically for security purposes. If the DNS name is set up, then we do not need to change the endpoint in our automation even if we spin up new instances and delete old ones. Secondly, the Wiremock ELB and Wiremock instance are independent from the SUT set up. We can deploy a SUT without bringing down the Wiremock instance and losing the stubs that were previously set. If we do not want the Wiremock server anymore, we just need to simply point the SUT's endpoint back to the DOC and delete the Wiremock instance and Wiremock ELB.

In this set up, the flow is: The automation test calls the Wiremock ELB, which forwards the request to the Wiremock EC2 instance to set up the stubs. Then the automation test calls the system under test ELB that proxies the request to the SUT instance. The SUT calls the Wiremock ELB. Wiremock ELB forwards the requests to the Wiremock EC2 instance. The Wiremock EC2 instance returns the stubbed response back to the Wiremock ELB and goes back to the automation test through the same route.

At the time we set up stubs, we also specify the request filter. Wiremock will only return the stubs if the request matches the filter. If it does not match, it will forward the request to the DOC and relay the response back.

4.1.3 Stub Requests

All the Wiremock code in this paper uses Rest-assured [7], a web service testing framework, to do the Wiremock API call.

This is the command line call to start the Wiremock server.

java -jar wiremock-1.53-standalone.jar --verbose --port 8080 --proxy-all=[DOC DNS Name]

This sets the Wiremock to return with status equals to 200 and the body we specified only when the endpoint equals to “/from/where”. “get(urlEqualTo(“/from/where”))” is the request filter for this stub.

```
stubFor(get(urlEqualTo(“/from/where”))
                .willReturn(aResponse().withStatus(200)
                .withHeader("Cache-Control", "no-cache")
                .withHeader("Content-Type", "application/json")
                .withBody(“Taiwan” )));
```

This calls the Wiremock API to do fault injection for the specified endpoint.

```
stubFor(get(urlEqualTo(“/some/thing”))
                .willReturn(aResponse()
                .withHeader("Cache-Control", "no-cache")
                .withFault(Fault.EMPTY_RESPONSE)));
```

4.2 Automate Setting Up Wiremock in AWS

4.2.1 Automate Launching the ELB and EC2

Potentially you can manually create the ELB and EC2 and just leave them there. However, some companies have policies to regularly clean up the AWS resources for security purposes. Then it is essential to automate these otherwise time-consuming and tedious manual steps. Automating them saves a lot of time, eliminates possible human errors and allows other teams to reuse them. Another major advantage is that you can put the automation script under revision control and review different versions if things ever go wrong.

We used AWS CloudFormation to automate launching ELB and EC2. CloudFormation is a service that helps the user to model and set up AWS resources easily [8]. It is provided by AWS and is widely adopted by the AWS user community. In addition, our team was already using it for application server deployment. With those reasons, we decided to choose CloudFormation. We created a CloudFormation template for the ELB and EC2 that specifies all the parameters that are required to launch the resources, i.e. instance type, security group, IAM roles and other parameters. The templates are in JSON format. You can also use CloudFormer to create CloudFormation templates from existing AWS resources. After the templates were created, we then called the CloudFormation API with the template in the payload to launch the AWS resources. It is simple and elegant.

4.2.2 Automate Deploying Wiremock

Deploying Wiremock requires downloading the jar into the EC2 instance and starting the Wiremock server. While you can do so manually, automating this allows other teams to reuse it and you can check in the code and track different versions. At Intuit, we chose to use Chef, a widely popular infrastructure automation framework. Chef turns infrastructure into code. You can code how you build, deploy and manage your infrastructure [9]. For example, you can write a Chef web server recipe to instruct Chef to download and install apache and JDK, run shell scripts to start the server, set up ports and users on the target server, etc.

In the Chef repo, we used Berkshelf, a dependency manager for Chef, to pull all the required recipes [10]. For our Berkshelf, it pulls the Java recipe and Wiremock recipe from our Intuit repository. Java recipe installs Java on the EC2 instance. The Wiremock recipe downloads the Wiremock jar from the maven and executes the command to start the Wiremock server. We extracted several parameters into attributes that are common to the cookbook. This allows using the same cookbook to deploy Wiremock to different pre-production environments. Extracting the parameters common to different environments into attributes ensures that they are consistent when deploying to the different environments. One example will be the Wiremock jar version. We want it to be the same for all environments.

5. Conclusion

With stubs and proxies in the AWS, we are able to test different permutations of responses from depended-on components easily in the test automation. It not only significantly increases our code coverage, makes our system under test more robust, but also allows teams to develop in parallel, move faster and be agile without being blocked by a DOC. Moreover, the fault injection functionality provided by the Wiremock allows us to do resiliency automation testing with a simple API call. We are able to catch bugs before they escape into production. It is hard to achieve before. Doing stubs and resiliency testing in AWS poses some challenges, including the high availability architecture and learning curve for AWS API and AWS operating environment. But it also presents unique opportunities to empower the team, since we now have full control over our infrastructure management. We can delete the instances whenever we want and provision new instances and deploy quickly for a fresh environment. With all these combined, the code quality is significantly increased and fewer bugs are introduced to production.

References

[1] "Depended-on Component (DOC)." *DOC at XUnitPatterns.com*. Web. 28 July 2015. <http://xunitpatterns.com/DOC.html>.

[2] "Test Stub." *At XUnitPatterns.com*. Web. 28 July 2015. <http://xunitpatterns.com/Test Stub.html>.

[3] *Wikipedia*. Wikimedia Foundation. Web. 28 July 2015. <https://en.wikipedia.org/wiki/Fault_injection>.

[4] "Resiliency Testing." *The IT Law Wiki*. Web. 28 July 2015. <http://itlaw.wikia.com/wiki/Resiliency_testing>.

[5] "VPC and Subnets." *Amazon Virtual Private Cloud*. Web. 28 July 2015. <http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_Subnets.html>.

[6] "WireMock." *WireMock*. Web. 28 July 2015. <http://www.wiremock.org>.

[7] "Rest-assured." *GitHub*. Web. 28 July 2015. <https://code.google.com/p/rest-assured>.

[8] "AWS CloudFormation." *AWS CloudFormation*. Web. 28 July 2015. <http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html>.

[9] "Chef." *Chef*. Web. 28 July 2015. <https://www.chef.io/chef/>.

[10] "Berkshelf." *Berkshelf*. Web. 28 July 2015. <http://berkshelf.com/>.