Extracted from:

Scalable Cloud Ops with Fugue

Declare, Deploy, and Automate the Cloud

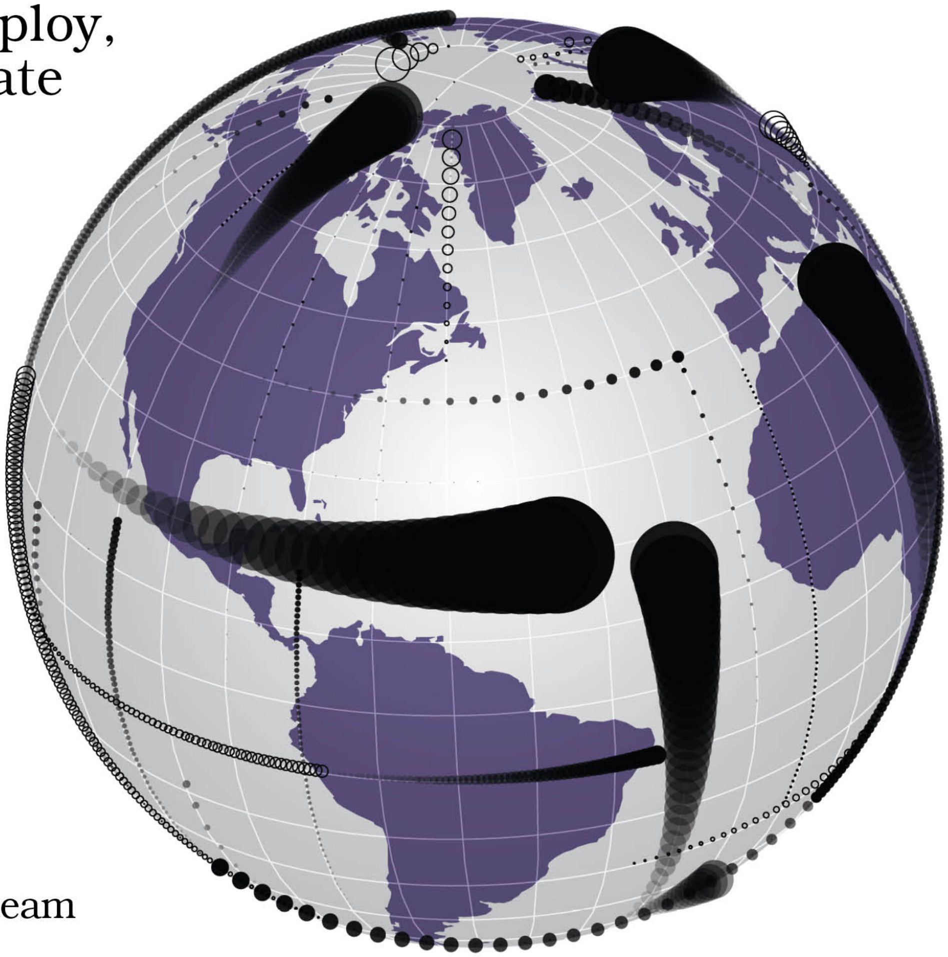
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Scalable Cloud Ops with Fugue

Declare, Deploy, and Automate the Cloud



Josha Stella
and the Fugue team

Scalable Cloud Ops with Fugue

Declare, Deploy, and Automate the Cloud

Josha Stella and the Fugue team

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Liz Welch
Layout: Gilson Graphics

Printed in the United States of America.
ISBN-13: 978-1-68050-234-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2017

Reducing Redundancy in Refuge ASGs and Beyond

Let's apply more of what you've learned to our Refuge composition. Looking at the code, we can see that there are many similarities between the refuge-web-app, refuge-api, and refuge-notifiication-worker services. Namely, each is composed of the following resources:

- AutoScalingGroup
- LaunchConfiguration
- SecurityGroup
- InstanceProfile (and associated role and policy)

A few differences aside, all of these resources are configured in pretty much the same way. Let's pull that common structure out to a function, both to reduce duplication and to allow us to focus on the interesting parts of our services, rather than the boilerplate they have in common. We'll use an approach similar to the loadbalancer function we wrote earlier in *Tightening Refuge Code with a Function*, on page ?, but this time our resources differ in more than just their names.

Defining a Function to Return an ASG

The first step is to identify where the services differ. One obvious difference is that the refuge-web-app and refuge-api services are both behind a LoadBalancer, while the refuge-notification-worker service is not. Also notice that the refuge-api and refuge-notification-worker services specify a user-data script, while the refuge-web-app service does not. Other differences include the AMI instances' boot, the SSH key pair, and the IAM policy applied to the service's instances. We'll need to capture each of these differences in a function parameter, which could be done using positional arguments, but given the number of arguments, that would probably be tough to keep straight over time. So, we'll use named arguments.

The function header would then look like this:

```
fun service {
      name: String,
      image: String,
      policyFile: String,
      keyName: String,
      loadBalancer: Optional<ELB.LoadBalancer>,
      userData: Optional<String>
    } -> AutoScaling.AutoScalingGroup:
```

We defined the function to return an AutoScalingGroup, which will contain references to the other resources required for a service. We have four required arguments:

```
name
image
policyFile
keyName
```

and two optional arguments:

```
loadBalancer
userData
```

The required arguments are pretty much what you would expect: name will be used to generate meaningful names for the resources we create as part of our service and image is the AMI we'll be booting. policyFile is the path to the external file containing the IAM policy's JSON. We'll be reading this file from the disk and using it to generate an IAM.Policy, an IAM.Role, and an IAM.InstanceProfile for the service. keyName is the name of the SSH key pair to configure on the instances.

Our first optional argument is loadBalancer, which allows us to pass in a LoadBalancer to put in front of our service's instances. We'll also configure the service's SecurityGroup to allow connections from the LoadBalancer on port 80. If the service doesn't use a LoadBalancer, we'll leave the SecurityGroup as is, which matches the refuge-notification-worker configuration. Finally, we have the optional userData, which is used to specify the user-data script to configure for instances.

This definition will allow us to define services using the following syntax:

```
refuge-web-app: service {
  name: "refuge-web-app",
  image: refuge-web-app-ami,
  policyFile: "policies/EC2.json",
  loadBalancer: refuge-web-app-elb,
  keyName: "book-app-us-west-2"
}

refuge-notification-workers: service {
  name: "refuge-notification-workers",
  image: refuge-notification-worker-ami,
  policyFile: "policies/worker_policy.json",
  keyName: "refuge-notification-us-west-2",
  userData: "#! /bin/bash
\export SQS_QUEUE=refuge-event-queue
\export SNS_TOPIC=refuge-notification-topic"
}
```

This is considerably shorter and more readable than the previous definitions, which are spread over many top-level bindings.

Implementing the Function

Now that you have an idea of what the service function will look like, let's start implementing it. As mentioned, we'll be returning a single AutoScalingGroup, so we'll need to create local bindings for the IAM Policy, Role, InstanceProfile, SecurityGroup, and LaunchConfiguration and then use those intermediate values to assemble the final AutoScalingGroup.

Start with the IAM resources, they're the simplest and depend only on policyFile:

```
let policy: IAM.Policy.new {
  policyName: name ++ "-policy",
  policyDocument: String.readFileUtf8(policyFile),
}
let role: IAM.Role.new {
  roleName: name ++ "-role",
  assumeRolePolicyDocument: IAM.Policy.AssumeRole.ec2,
  rolePolicies: [policy],
}
let profile: IAM.InstanceProfile.new {
  instanceProfileName: name ++ "-profile",
  roles: [role],
}
```

Here we've created the policy, role, and profile local bindings. The only difference from the previous definitions is that we're building the resource names based on the service name (so the instance profile for refuge-web-app is named "refuge-web-app-profile").

Next, let's create our service's security group. This is a little more involved, as we need to construct the rules based on the presence of a load balancer:

```
let sg: EC2.SecurityGroup.new {
  description: name,
  vpc: refuge-network.vpc,
  ipPermissions:
    case loadBalancer of
      | None         -> []
      | Optional elb ->
        let elbSgs: Optional.unpack(
          [],
          elb.(ELB.LoadBalancer).securityGroups,
        )
        [
          EC2.IpPermission.http(
            EC2.IpPermission.Target.securityGroups(elbSgs)
          )
        ]
}
```

The description and vpc arguments are straightforward, but let's dig into what's going on with ipPermissions. The goal is to create a security group with no rules if our service doesn't use a load balancer, which is accomplished by pattern-matching on the loadBalancer argument and returning the empty list—that is, None—if no load balancer is provided.

```
case loadBalancer of
  | None -> []
```

When we provide our service with a load balancer, we need to extract its security groups and create a rule allowing access to port 80 from the load balancer's security groups. To do that, we first have to unpack the underlying ELB.LoadBalancer from the Optional value using pattern matching. Once we have an ELB.LoadBalancer, we can extract the security groups:

```
elb.(ELB.LoadBalancer).securityGroups
```

ELB.LoadBalancer's securityGroups field is of type Optional<List<EC2.SecurityGroup>>, which means we'll need to unpack the list from an Optional. We need to provide a default value of [] since we're expecting a List<EC2.SecurityGroup>:

```
let elbSgs: Optional.unpack(
  [],
  elb.(ELB.LoadBalancer).securityGroups
)
```

We now have a List<EC2.SecurityGroup>, which we can use with the EC2.IpPermission.Target.securityGroup function to create a single rule that allows access from the security groups in the list:

```
[
  EC2.IpPermission.http(
    EC2.IpPermission.Target.securityGroups(elbSgs)
  )
]
```

The rule is wrapped in a list and passed to EC2.SecurityGroup.new as the ipPermissions argument, giving us our security group.

Next up is combining the previously created values into a LaunchConfiguration:

```
let lc: AutoScaling.LaunchConfiguration.new {
  image: image,
  securityGroups: [sg],
  instanceType: EC2.T2_micro,
  associatePublicIpAddress: if Optional.isNone(loadBalancer)
                            then False
                            else True,
  iamInstanceProfile: profile,
  keyName: keyName,
  userData: userData,
}
```

We want to give our service instances public IP addresses only if they're hooked up to a load balancer. LaunchConfiguration.new's associatePublicIpAddress argument is a Bool, so we'll need to convert our Optional<ELB.LoadBalancer> argument into a Bool.

```
if Optional.isNone(loadBalancer)
then False
else True
```

Optional.isNone is a standard library function that returns True if the given optional value is None, and returns False otherwise. We could also do this test with pattern matching:

```
case loadBalancer of
  | None       -> False
  | Optional _ -> True
```

But in this case, we're not doing anything with the Optional value and the equivalent if/then/else expression is shorter.

Finally, we need to assemble and return our AutoScalingGroup:

```
AutoScaling.AutoScalingGroup.new {
  subnets: refuge-network.publicSubnets,
  minSize: 2,
  maxSize: 2,
  defaultCooldown: 300,
  healthCheckType: AutoScaling.Ec2,
  launchConfiguration: lc,
  tags: [refuge-tag],
  terminationPolicies: [AutoScaling.ClosestToNextInstanceHour],
  enabledMetrics: [
    AutoScaling.GroupInServiceInstances,
    AutoScaling.GroupTotalInstances
  ],
  loadBalancers: case loadBalancer of
                 | None         -> None
                 | Optional elb -> Optional([elb])
}
```

The only manipulation we need to do here is to convert our loadBalancer argument from an Optional<ELB.LoadBalancer> to an Optional<List<ELB.LoadBalancer>>, which we do via pattern matching.

```
case loadBalancer of
  | None          -> None
  | Optional elb -> Optional([elb])
```

Note that we cannot rely on optional lifting here (recall *Optional Arguments*, on page ?), so we must explicitly wrap the list in an Optional.

Refactoring the Function to Problem-Solve

At this point our function is complete and we're able to generate AutoScalingGroups and all of the associated resources for our services. However, we run into one snag when trying to refactor our composition to use a service—that is, we reference the security groups for the refuge-web-app and refuge-api services in the security group rules for our RDS database instance:

```
refuge-rds-sg: EC2.SecurityGroup.new {
  description: "Allow MySQL traffic from the Internet",
  ipPermissions: [
    EC2.IpPermission.mysql(
      EC2.IpPermission.Target.securityGroup(refuge-web-app-sg)
    ),
    EC2.IpPermission.mysql(
      EC2.IpPermission.Target.securityGroup(refuge-api-sg)
    ),
  ],
  tags: [refuge-tag],
  vpc: refuge-network.vpc
}
```

We could extract the security groups from the service AutoScalingGroups, but there's a cleaner way to solve the problem with a small change to our service function. Now that we know we need access to both the Auto Scaling group and the security group, we can introduce a new type that contains both values in a way that's convenient to access in our composition:

```
type Service:
  asg: AutoScaling.AutoScalingGroup
  securityGroup: EC2.SecurityGroup
```

We can then tweak the service definition to return values of type Service, after which we'll be able to access the security group using the normal dot notation:

```
refuge-rds-sg: EC2.SecurityGroup.new {
  description: "Allow MySQL traffic from the Internet",
  ipPermissions: [
    EC2.IpPermission.mysql(
      EC2.IpPermission.Target.securityGroup(
        refuge-web-app.securityGroup
      )
    ),
    EC2.IpPermission.mysql(
      EC2.IpPermission.Target.securityGroup(
        refuge-api.securityGroup
      )
    ),
  ],
  tags: [refuge-tag],
  vpc: refuge-network.vpc
}
```

Let's make the changes to service. First, we'll update our function's type to reflect the new return value:

```
fun service {
      name: String,
      image: String,
      policyFile: String,
      keyName: String,
      loadBalancer: Optional<ELB.LoadBalancer>,
      userData: Optional<String>
    } -> Service:
```

Then we need to update the function's body to create a Service record instead of returning the AutoScalingGroup directly. We'll leave the IAM policy, security group, and launch configuration local bindings as they are.

```
let asg: AutoScaling.AutoScalingGroup.new {
  subnets: refuge-network.publicSubnets,
  minSize: 2,
  maxSize: 2,
  defaultCooldown: 300,
  healthCheckType: AutoScaling.Ec2,
  launchConfiguration: lc,
  tags: [refuge-tag],
  terminationPolicies: [AutoScaling.ClosestToNextInstanceHour],
  enabledMetrics: [
    AutoScaling.GroupInServiceInstances,
    AutoScaling.GroupTotalInstances
  ],
  loadBalancers: case loadBalancer of
                 | None         -> None
                 | Optional elb -> Optional([elb])
}
{asg: asg, securityGroup: sg}
```

This is pretty much the same as our original implementation, with a new local binding to hold the AutoScalingGroup and the return value converted to be a record literal of type Service.

We're now able to convert our composition over to using our new service function and remove the corresponding top-level resources. In addition to removing some code, we're in a good position to evolve our composition. Adding a new service is just a couple of lines of code, and modifying our service configuration is standardized and localized to a relatively small chunk of code.