Extracted from:

Scalable Cloud Ops with Fugue

Declare, Deploy, and Automate the Cloud

This PDF file contains pages extracted from *Scalable Cloud Ops with Fugue*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2017 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

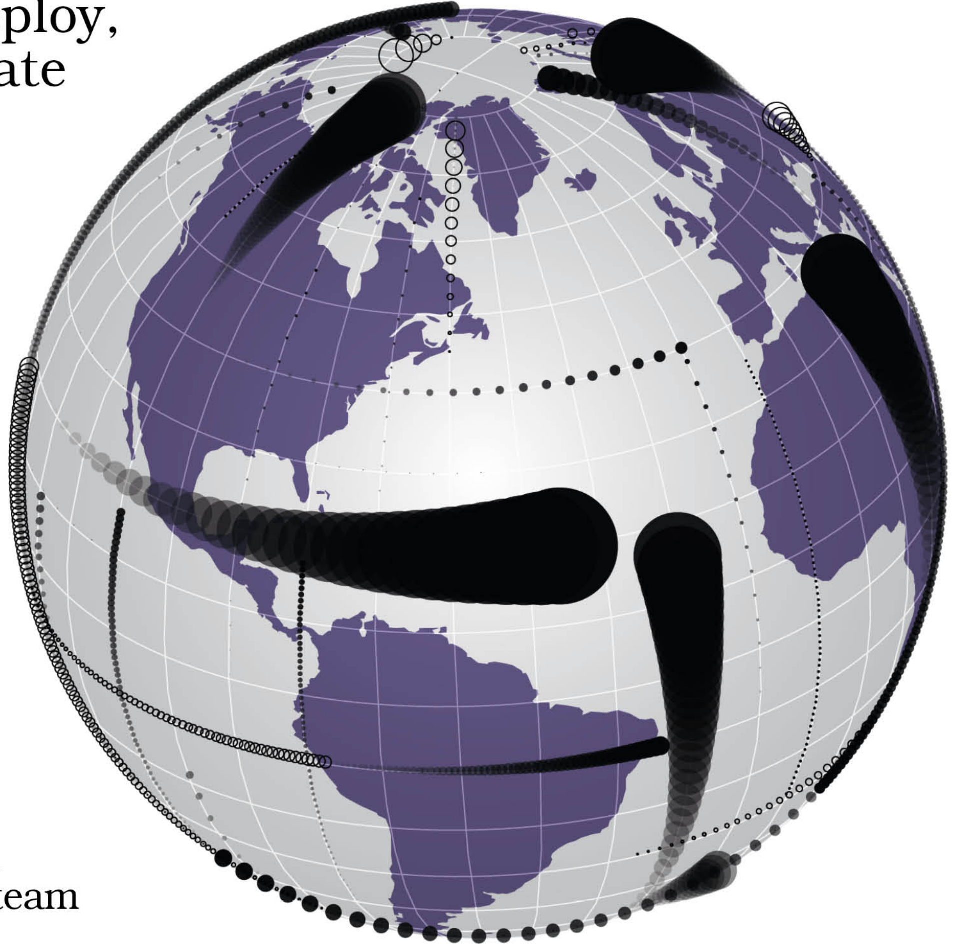
The Pragmatic Bookshelf

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2017 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

Composing Refuge's Core Components

In the Fugue Ink example in the first chapter, you learned that Ludwig is a powerful, statically typed, domain-specific language that is the programming interface to Fugue. As we write the Refuge composition, you'll learn how to define multiple application components using the Ludwig types. You'll also use the formative capabilities present in those Ludwig types to quickly and cleanly define the network and security connections that exist between the application components.

Keep in mind that the Refuge application, like Fugue Ink, will use AWS as the infrastructure cloud provider. Following is a quick mapping of various Refuge application components to specific AWS services:

- Load balancers: AWS Elastic Load Balancer (ELB)
- Web servers: Amazon Elastic Compute Cloud (EC2)
- API servers: EC2
- Firewall/network rules: AWS security groups (SGs)
- MySQL database: Amazon Relational Database Service (RDS)
- Scaling: AWS Auto Scaling groups (ASG)

We're all used to writing services and then having their configuration for infrastructure and communications scattered across many interfaces, such as firewall rules, configuration files, and management interfaces. With Fugue, the complete implementation of your service—from network connections to shared variables to scaling and performance requirements—can be succinctly declared in one place: the composition. Let's start writing.

Kicking Off the Composition

In spinning up the Fugue Ink application, you've already installed Fugue and Ludwig on your machine. (If you don't have the files, just follow the instructions in *Installing Fugue and Ludwig*, on page ?.) Next, create a new file. Call it RefugeChapter3.lw and add the following one-word line at the beginning of the composition, just as you did for FugueInk.lw in *A Simple Composition*, on page ?.

RefugeChapter3.lw

```
composition
```

Notice that if you click on the highlighted link for RefugeChapter3.lw in the digital book, you'll access the entire file for this chapter. You can also access this file

from the book's companion website.[1] But build along with us, step-by-step, to benefit from the explanations if you'd like. There isn't much going on here right now with a single line. We've simply initiated a new composition by creating the new file and using this declaration. Including this line is crucial, however, because it signals to the Conductor that this is a composition file and it can be executed. A Ludwig file that does not contain this line will be viewed as a library, which can be included in other compositions but cannot be executed alone. We'll use libraries in every composition, but will deep-dive into the concept and mechanics later, in Chapter 6, *Understanding Ludwig*, on page ?, and Chapter 7, *Ludwig Modules and Validations*, on page ?.

Importing Libraries

Let's import some libraries to get us started. The Fugue client you installed on your computer ships with the Fugue Standard Library and you'll frequently reference modules from this library. It contains three different types of modules: Fugue.Core.*, Fugue.AWS.*, and Fugue.AWS.Pattern.*.

Fugue.Core.*

The Fugue.Core.* namespace contains the lowest-level modules that interact directly with the AWS CLI. They're actually used in building the next two types of modules described. They tend to have little validation, and you'll use these only if there is no equivalent, higher-level module available. Statements like the following support this low-level AWS functionality; we'll use the Vars in our app for the time being:

```
import Fugue.Core.AWS.Common as AWS
import Fugue.Core.Vars as Vars
```

Fugue.AWS.*

The Fugue.AWS.* namespace contains a module for each AWS service. (At the time of this writing, Fugue provides coverage for many AWS services, with an aim to have complete coverage in the near future.) These modules allow you to declare your infrastructure by defining fields that roughly correspond to the AWS CLI, but they provide some additional validation and type-checking.

Let's import modules for the AWS services that we plan on using:

```
import Fugue.AWS as AWS
import Fugue.AWS.EC2 as EC2
import Fugue.AWS.ELB as ELB
import Fugue.AWS.RDS as RDS
```

1. https://pragprog.com/book/fugue

```
import Fugue.AWS.AutoScaling as AutoScaling
import Fugue.AWS.IAM as IAM
```

Fugue.AWS.Pattern.*

The Fugue.AWS.Pattern.* namespace contains modules for common cloud architecture patterns, written by Fugue architects. Under the hood, these modules are written using the same Fugue.AWS service modules that we just imported, but they add a useful level of abstraction. In our application, we need to add some network configuration to ensure the EC2 instances and the RDS database are able to securely communicate with each other and with the Internet, so we'll import the Network pattern module:

```
import Fugue.AWS.Pattern.Network as Network
```

Because you have access to the underlying AWS service modules that the network pattern was built with, you can build your own libraries in exactly the same way. This is typically useful in cases where you're declaring the same kind of infrastructure over and over again, and it can keep your code more concise.

Defining Resources

Now that we've initialized our composition with a name and imported libraries, we can start building our infrastructure. Let's begin by defining a tag that we'll add to all of our AWS resources. Tags are just pieces of metadata that you can use to help manage your infrastructure. For example, you can view EC2 instances by tag in the AWS Management Console. (Remember, for brevity, we'll call it the AWS Console.) For clarity, we'll tag every infrastructure component of our project with this application tag:

```
# Misc
refuge-tag: AWS.Tag {
  key: "Application",
  value: "Refuge"
}
```

As you can see, we just created a variable called refuge-tag and bound a new tag to it. Let's also create a refuge-region variable for the region into which we'll be deploying our application, since that's a value that might be used multiple times in our composition:

```
refuge-region: AWS.Us-west-2
```

Now we need to create the basic infrastructure required to get Refuge up and running: a secure network, compute resources to run the Refuge application code, and a database.

Creating a Network

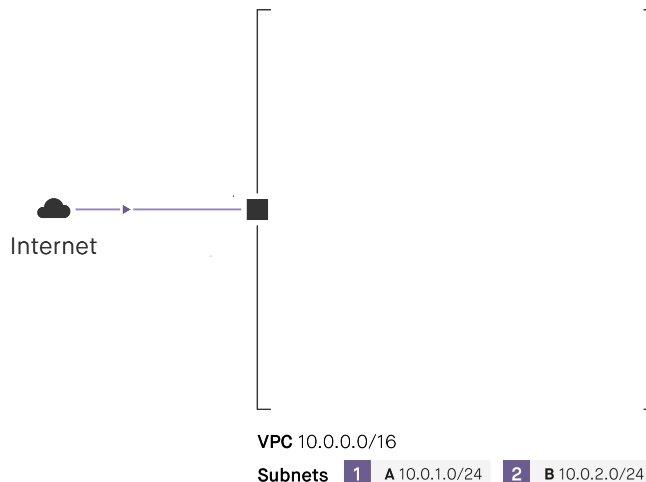
Before we create the instances that will run our application, we need to define a secure network for them. As you know from Chapter 1, *Simple Themes and First Steps*, on page ?, Virtual Private Cloud (VPC) is AWS's answer to the traditional data center network. Remember that VPCs are isolated networks in AWS. You can configure your VPC's IP address range, route tables, and security settings.

Recall, too, from the first chapter that the `Network` pattern is a Fugue abstraction that defines a VPC, subnets, an Internet gateway, and a routing table. For simplicity's sake, we'll stick to a VPC with public subnets. Later we may choose to harden our security and create some private subnets for our databases.

• Click HERE to purchase this book now. discuss

```
# Network
refuge-network: Network.public {
  name: "Refuge Network",
  region: refuge-region,
  cidr: "10.0.0.0/16",
  subnets: [
    (AWS.A, "10.0.1.0/24"),
    (AWS.B, "10.0.2.0/24"),
  ]
}
```

In a small, concise block of code, we've accomplished all of the complex network configuration we need to get our application running in a secure, isolated network. You see a simple visualization of this in the following diagram, which corresponds to our Ludwig declarations and which is the starting point as we grow our application's infrastructure throughout this chapter.



Creating Compute Resources

Refuge is made up of two separate applications—the web front end and the API—so we need two distinct sets of compute resources, in this case EC2 instances. Refuge is not sure how popular its social network will be right off the bat, so we'll build in some basic scaling and load balancing from the start, just in case.

Instead of creating individual EC2 instances like you did for the Fugue Ink application in Chapter 1, *Simple Themes and First Steps*, on page ?, you'll create an Auto Scaling group (ASG) for each of the two applications. An ASG is a way to impart basic autoscaling behavior to a service running on AWS. It can scale EC2 instances up or scale down (within limits) and take performance metrics as input. It ensures that an adequate number of healthy instances are available to serve the application. We'll configure both ASGs to have a minimum and a maximum of two instances, so we will have a total of four EC2 instances running at all times.

To balance traffic between these instances, we'll use an ELB, AWS's virtualized load balancer. These are used in front of a collection of EC2 instances to balance traffic across them. Using an ELB will improve the fault tolerance of our application by routing traffic to only healthy instances. We can add more instances to an ELB while it's running, so if we decide to update our composition later with higher instance limits on our ASG, the new instances can be added to the ELB. We'll configure two ELBs, one for each ASG. (Note that Application Load Balancers—ALBs—are an ELB option appropriate to some architectures; we'll stick with the classic ELB in our example.[2])

Incoming traffic from the Internet will go to the ELB, which will, in turn, route traffic to an instance. For this to happen securely, we need to define security groups to limit access for ELBs and EC2 instances:

```
# ELB Security Group
refuge-elb-sg: EC2.SecurityGroup.new {
  description: "Refuge loadbalancer",
  ipPermissions: [
    EC2.IpPermission.http(EC2.IpPermission.Target.all),
    EC2.IpPermission.https(EC2.IpPermission.Target.all),
  ],
  tags: [refuge-tag],
  vpc: refuge-network.vpc
}

# EC2 Security Group
refuge-ec2-sg: EC2.SecurityGroup.new {
  description: "Allow http traffic from the ELB",
  ipPermissions: [
    EC2.IpPermission.http(EC2.IpPermission.Target.securityGroup(refuge-elb-sg)),
  ],
  tags: [refuge-tag],
  vpc: refuge-network.vpc
}
```

2. https://aws.amazon.com/elasticloadbalancing/

The refuge-elb-sg security group allows HTTP and HTTPS traffic to flow from the Internet to the ELB. The refuge-ec2-sg security group limits EC2 instance traffic exclusively to HTTP traffic coming from the ELB.

Now that the security rules are taken care of, we'll define our two ELBs:

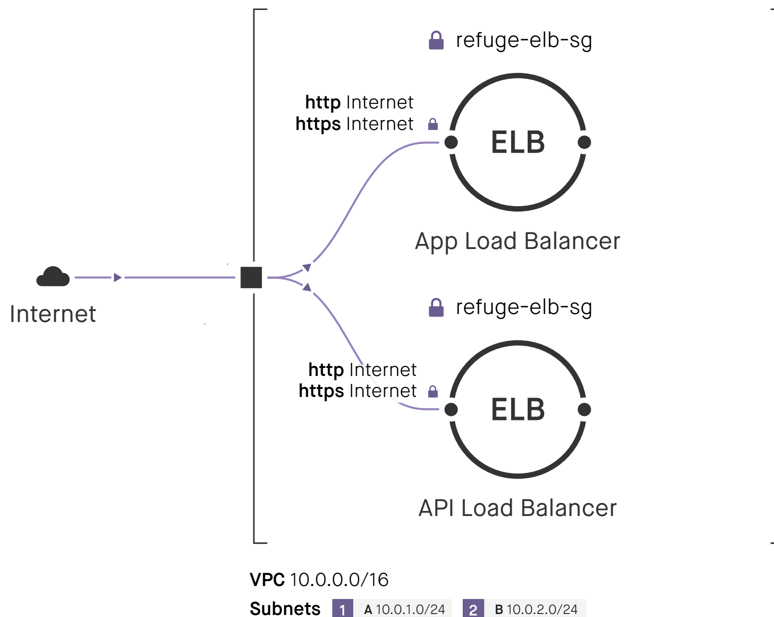
```
# Web App ELB
refuge-web-app-elb: ELB.LoadBalancer.new {
  loadBalancerName: "refuge-web-app",
  healthCheck: refuge-elb-health-check,
  subnets: refuge-network.publicSubnets,
  securityGroups: [refuge-elb-sg],
  listeners: [refuge-http-listener],
  tags: [refuge-tag]
}

# API ELB
refuge-api-elb: ELB.LoadBalancer.new {
  loadBalancerName: "refuge-api",
  healthCheck: refuge-elb-health-check,
  subnets: refuge-network.publicSubnets,
  securityGroups: [refuge-elb-sg],
  listeners: [refuge-http-listener],
  tags: [refuge-tag]
}

# ELB health check
refuge-elb-health-check: ELB.HealthCheck.tcp {
  interval: 30,
  timeout: 5,
  unhealthyThreshold: 3,
  healthyThreshold: 2,
  port: 80
}

# ELB HTTP listener
refuge-http-listener: ELB.Listener.new {
  protocol: ELB.HTTP,
  loadBalancerPort: 80,
  instancePort: 80
}
```

Note that many Ludwig AWS modules come with a default record that contains a default configuration for the service. Remember, defaults are applied automatically if you omit an argument.

We're only defining an HTTP listener here, so the application won't actually work over HTTPS. We would need to deploy an SSL server certificate to the ELB to enable HTTPS encryption and decryption. We can add this (extremely important) feature later in the production environment.

• Click HERE to purchase this book now. discuss

The following diagram shows that the composition now includes ELBs for each of our two ASGs and the corresponding security group for those ELBs.



The security group for our EC2 instances is not detailed in this diagram, but is logically grouped in the diagram shown on page ?.

Now we need to create our two ASGs. Each ASG has a launch configuration that specifies details about the instances it will create, including an Amazon Machine Image (AMI) to launch from. We've provided two AMIs with the Refuge web app and API code ready to go. AMIs are region-specific, so they'll work only in your account in the `us-west-2` region. These particular AMIs contain only limited Refuge functionality since it is a stubbed application for the purpose of this example. In reality, you'd update the AMIs to new ones with additional capabilities and more production-ready features as development proceeds. So, at the top of the composition you create variables that will be easy to change:

• Click HERE to purchase this book now. discuss