Extracted from:

Scalable Cloud Ops with Fugue

Declare, Deploy, and Automate the Cloud

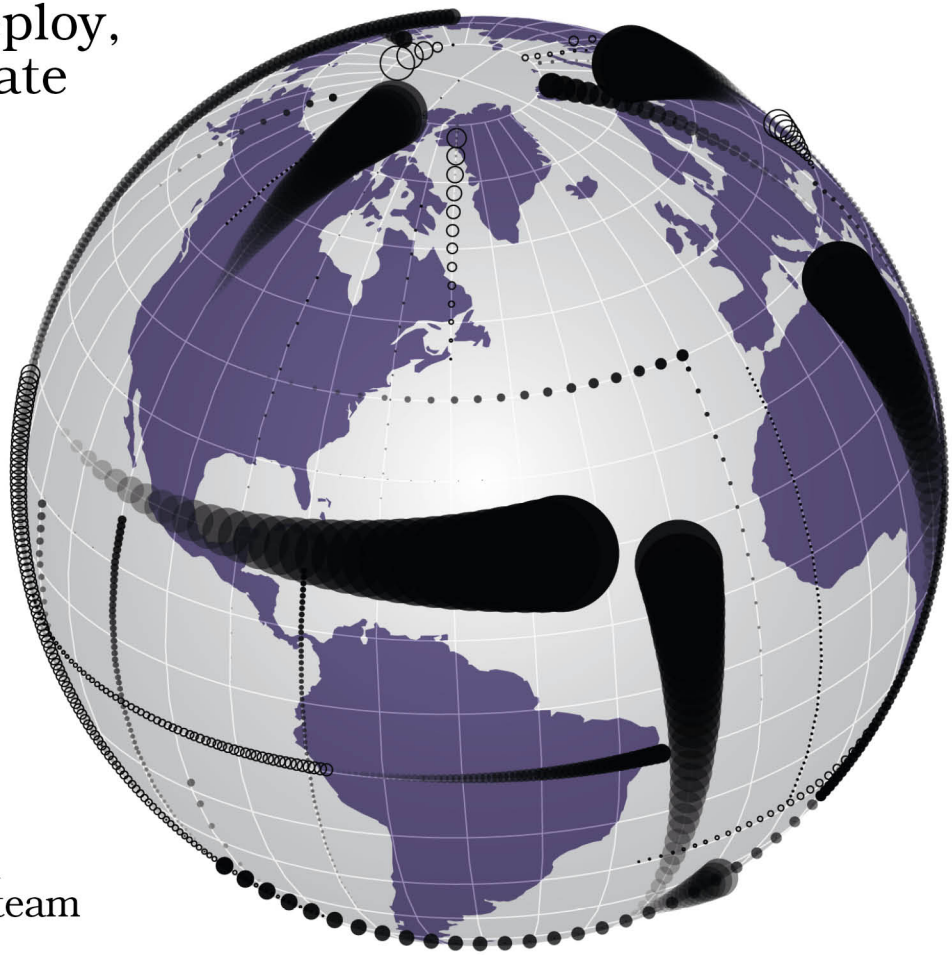
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Scalable Cloud Ops with Fugue

Declare, Deploy, and Automate the Cloud



Josha Stella
and the Fugue team

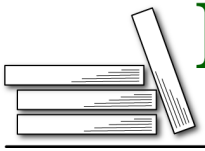
Scalable Cloud Ops with Fugue

Declare, Deploy, and Automate the Cloud

Josha Stella and the Fugue team

The Pragmatic Bookshelf

Raleigh, North Carolina



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*.

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

CHAPTER 8

Configuration and Coordination in Fugue: Vars

Throughout our story, the Refuge organization and its namesake application have grown. Each step of the way, we've deployed more robust versions of the application in the cloud with Fugue. You've coded its infrastructure in Ludwig—Fugue's simple but powerful language interface—increasing your expertise as you've added common, real-world services to the system in development and production.

Refuge, like most systems, maybe like the one you're working with in daily life, will experience growing pains and challenges as it scales from its first hundred to its first hundred thousand users. Some of these problems are purely application-level issues: How do we make sure the login system is secure? How do we deal with internationalization and time zones? How do we make sure users can find what they're looking for easily? And some of Refuge's problems are purely infrastructural: What instance type should we use for the web servers? Where can we host static assets? On what metrics should we horizontally scale each tier? These two types of problems—application and infrastructure—are thought of as disjointed sets of concerns and are often handled by different people.

As systems get larger and more complex, a third set of issues emerges, one that touches both the application and the infrastructure: How do we share or publish infrastructure configuration so that our application can use it? How do we pick a single instance of our application to perform some task? How do we make sure all instances of the application perform the same task, or agree on the state of the world? These problems are, in general, coordination problems. In this chapter, we'll look at Vars, a service for handling configuration and coordination with Fugue.

What Is Vars and How Does It Work?

Vars, an abbreviation of "variable service," is a replicated key/value store that has features and semantics that make it useful for coordination, configuration sharing, credential synchronization, and more.

Vars consists of a server-side component that runs on the Fugue Conductor's instance and a small, client-side binary called vars that you install and run on every instance or node that needs to publish or consume data from Vars. (Recall your introduction to the Conductor, Fugue's runtime engine, in Chapter 1, *Simple Themes and First Steps*, on page ?, and see Chapter 11, *What We Mean by a*, on page ?, for a full discussion.) The vars binary presents a local RESTful interface by which software on a node can perform operations against the data store, such as publishing a new value, reading a value, listing values, and deleting values. In addition to the RESTful interface, the vars binary can perform all operations on itself, making it very useful for shell scripting and other automations. Whenever a node publishes a key/value pair, Vars replicates it to every other Vars node. For example, a Refuge developer who is running vars on her laptop, as well as on all of the web servers, might publish refuge-prod-assets to the /static-asset-bucket key in Vars. Vars will replicate that key/value pair to all of the web servers, where the servers' software can use it—in this case, specifying the S3 bucket from which static assets are to be served.

Vars as a Service Registry

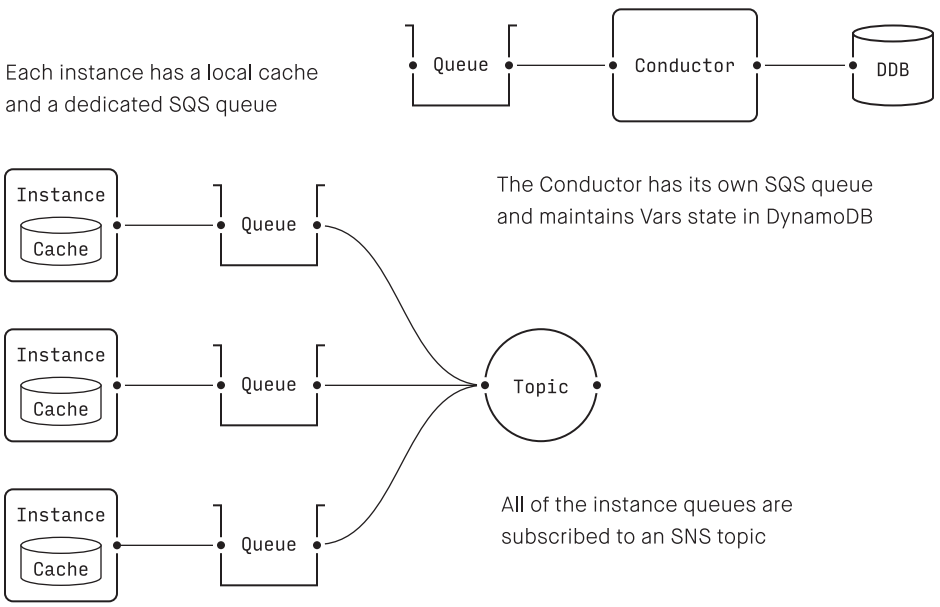
One of the core uses of Vars is as a service registry. To connect to a database, a web instance needs several pieces of configuration: the hostname or IP address for the database, the port on which the database is listening, a username and password with which to authenticate the communication, the default database name, and so on. *Where does the software on the web instance get this configuration?* We could hard-code these values into the web-instance machine image, but that makes configuration changes difficult in the future. We would need to have different images for each environment in which the image would run: for example, dev, test, and prod environments. Whenever we want to update the configuration for, say, a new database endpoint or because we have to rotate the database password, we would have to roll a new image and replace our entire fleet! We can solve this problem by using a service registry.

A service registry is a key/value store at a known location where pieces of configuration can be stored and shared. Database endpoints, CDN URLs, software version numbers, and cluster member lists are all examples of runtime configuration we can share with infrastructure components via a service registry.

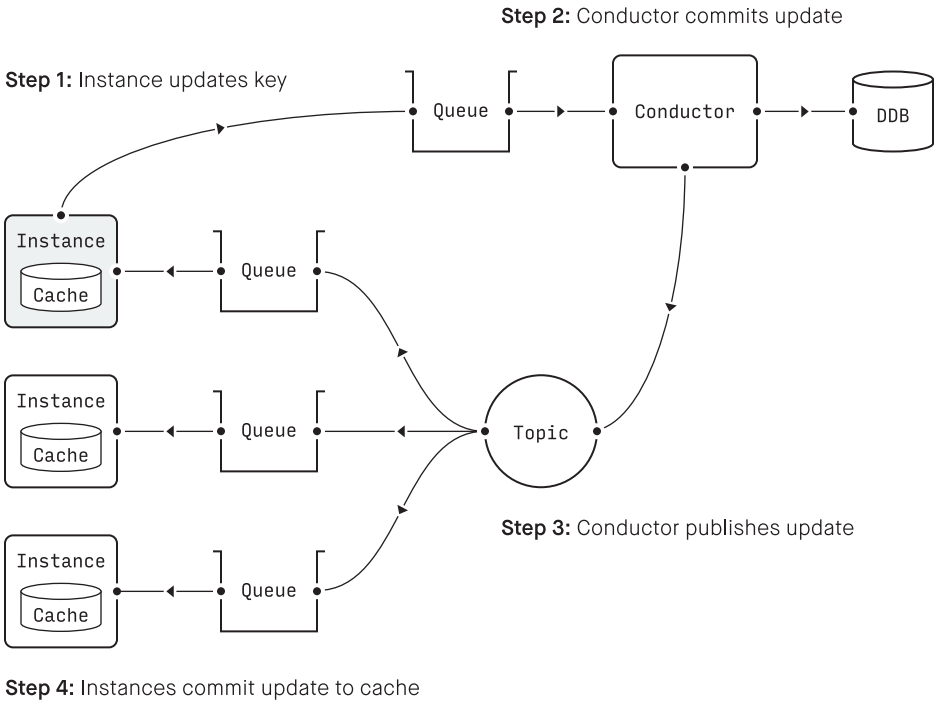
Operators or infrastructure components themselves can publish values to Vars that are usable by other components. With Fugue, when a database instance is created, the Conductor can publish the database hostname to Vars for the web instances to read and use. Also, an operator can publish database credentials (username and password) into Vars, and then the web-instance software can read those credentials and use them to connect to the database. We'll look at sharing sensitive information in *Sharing Refuge Passwords and Secrets with Fugue*, on page ?.

The Vars Workflow

Let's look at how Vars works. For every instance or node that is running Vars, Fugue's Conductor creates two SQS queues. One queue is used for replicating data between nodes, and the other is for receiving operations from the Conductor. This means that Vars does not require that instances have open ports for Vars to receive updates, which is great for security and makes network configuration easy. Each Vars instance also maintains a local cache with key/value pairs that it has received. The following diagram is a visual overview.



Whenever a Vars instance wants to publish a new value, that Vars instance sends an update message to the Conductor. This message contains the key/value pair, along with some metadata, like a version number. The Conductor receives the message and durably writes it to a log in DynamoDB. Once this update message has been stored, the Conductor publishes the updated data to every Vars instance via each one's SQS queue. When each instance receives the update, each writes it to its local cache. Note the write path in the diagram shown next.

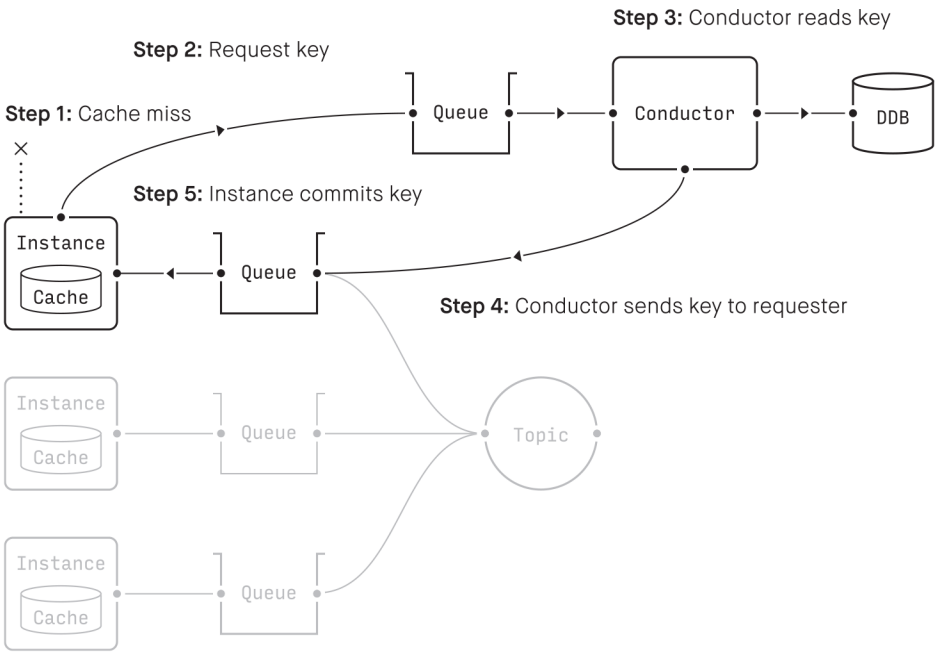


When a Vars instance is asked to fetch a value from the store, it first checks its local cache. Because Vars instances receive updates soon after they are committed, there is usually a high probability that the value is in the cache. In this case, the cached value is returned. In the case of a cache miss, the instance sends a message to the Conductor requesting the key/value pair. The Conductor fetches it from the DynamoDB log and publishes it to that instance, via its SQS queue. Again, the value is cached, so future reads are just local cache operations. See the read path on page 9.

A First Look at Optimistic Locking

When the Conductor writes an update to the log, it performs the write using optimistic locking. That is, the version number in the message is used to make sure that Vars nodes cannot overwrite stale versions of values. This is incredibly useful to deal with the situation where multiple nodes try to write to the same key at the same time. It's also useful for locking, which we'll cover in greater depth later in this chapter, in *Using Vars as a Lock Service*, on page ?.

Vars was designed to support a cluster of nodes with eventual consistency. An optimistic locking strategy is much more flexible and resilient in this type of environment. Since the vars binary running on each instance acts as a caching service, it is optimized to handle a high rate of read requests. If Vars were to use pessimistic locking, that would greatly reduce the efficiency of read operations, as they would be blocked and have to wait for any concurrent writes in the system to complete before returning information to a requestor. In a replicated keystore such as Vars, the implementation of optimistic locking is simpler, reducing the complexity of the system as a whole.



This design—where a Vars node effectively broadcasts an item of configuration or other data, which is then cached by other nodes—works well and is cost-effective for read-heavy workloads. For example, a single node publishing database connection parameters to a fleet of web servers constitutes a read-heavy workload.

Here, a node publishes the database connection parameters once, and then web servers read them many times. Extensive instance-side caching in Vars makes read operations very cheap, in terms of both performance and infrastructure costs.

Most of the costs associated with Vars, again with respect to performance and dollars, are in write operations. The most expensive component in the system is DynamoDB, where write operations are pricier than read operations. Because of the extensive caching in Vars as well as the lower cost of reads from DynamoDB in the event of a cache miss, Vars performs tasks efficiently and is inexpensive to run for read-heavy workloads, like the ones presented in this chapter. Vars will run just fine under write-heavy workloads, but it will be more expensive to operate.

Spinning Up Vars

In just a bit, we'll turn to a couple of fairly common and substantive use cases for you to peruse with our inquisitive Refuge team, but first let's cover some basics. We'll dive into details so that you can develop real facility with Vars and get a sense of its versatility. Keep the visuals of Vars architecture in mind as we examine its installation, commands, and API.

Installing Vars

Like the Fugue command-line utility, Vars is a statically linked binary, so it does not require any language runtimes or libraries to be installed. Vars comes with the Fugue CLI, so you already have it. If, however, in your infrastructure you were to create 20 EC2 instances and you want a Vars client on each one, it would be overkill to install the whole Fugue CLI on all the instances just to accomplish that. Instead, you would want to download the standalone Vars binary and install it on your EC2 instances. Do that from Fugue's Download Portal and choose the latest version of Vars for your platform.[1] We use the OS X package throughout this chapter.

Installation takes seconds. When it's completed, run the following commands in the shell or your favorite terminal emulator to launch Vars:

```
vars -vv daemon --region=<your_conductor_region>
```

The -vv and -v flags just give you different levels of verbosity in output. You can use either. If you've been running an older version, downloading the latest package will upgrade your client. When creating AMIs for use with Fugue,

1. https://download.fugue.co

such as the Refuge AMI, you should include Vars. All AMIs and containers in Fugue's Download Portal already include Vars, unless otherwise noted.

Learning Commands

One of the best ways to explore a new application, of course, is to start with a little help. Let's try that out with Vars and consider the available commands. Type this:

```
$ vars --help
```

You'll see a response like the following, showing your options in the CLI:

```
Usage:
  vars [OPTIONS] <command>

Application Options:
      --human    Human-readable text logging
      --host=    Client Port
  -v, --verbose  Enable verbose output/logging
      --version  Show the current version

Help Options:
  -h, --help     Show this help message

Available commands:
  daemon   Run the vars daemon
  delete   Delete values in vars
  get      Get a value from vars
  list     List keys in vars
  listget  Retrieve a list of values from vars
  put      Put a value into vars
  status   Retrieve the status of the vars daemon
```

We'll dig in a bit to each command, as facility with them is key to using Vars well and understanding its operational relationships.