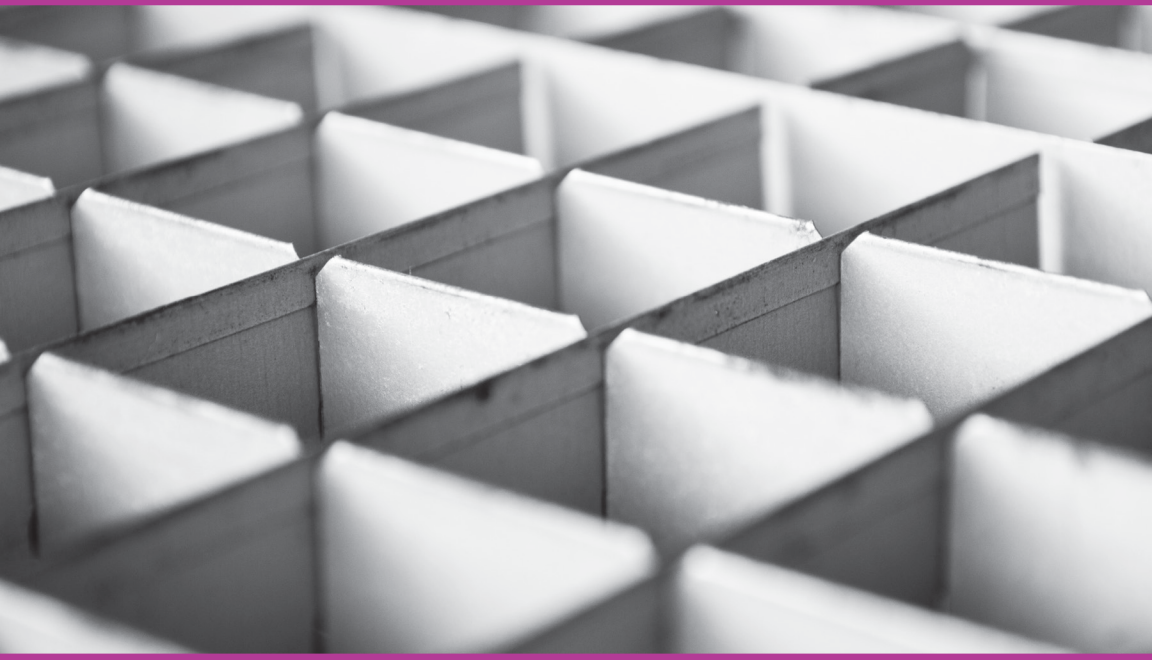
O'REILLY®

Compliments of Microsoft

How to Containerize Your Go Code



Liz Rice

How to Containerize Your Go Code

Liz Rice

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

How to Containerize Your Go Code
by Liz Rice

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Susan Conant
Production Editor: Nicholas Adams
Interior Designer: David Futato
Cover Designer: Randy Comer

March 2018: First Edition

Revision History for the First Edition
2018-02-28: First Release

This work is part of a collaboration between O'Reilly and Microsoft. See our *statement of editorial independence*.

978-1-492-03854-2

[LSI]

Table of Contents

How to Containerize Your Go Code

Introduction and Motivations

Learning about containers is a bit like learning about Linux or learning about Go: it's potentially a huge topic! But everyone has to begin somewhere. This lesson will give you an introduction to some of the key concepts of containers and walk you through some examples of using Docker containers with Go code.

Example Code

There is example code throughout this lesson. If you'd like to try it out for yourself, the easiest way is to download the code with by using the `go get` command, as follows:

```
$ go get github.com/lizrice/hello-container-world/...
```

Using Docker

If you've never used Docker before, good instructions are available for installing it and verifying that everything is set up correctly. After you've done that, you'll be ready to work through the examples in this lesson.

NOTE

Docker 1.13 reorganized the command-line interface (CLI) to refer to objects (e.g., `docker image build` instead of `docker build`). As of this writing, the older versions work as aliases, but it is more future-proof to get used to the new style.

What Is a Container?

Containers let you isolate an application so that it's under the impression it's running on its own private machine. In that sense, a container is similar to a virtual machine (VM), but it uses the operating system kernel on the host rather than having its own.

You start a container from a container image, which bundles up everything that the application needs to run, including all of its runtime dependencies. These images make for a convenient distribution package.

The isolation that fools a container into thinking it has control over an entire Linux machine is created by using namespaces and control groups. You don't need to know the details of these to use containers, but people tell me they find this talk I gave at Golang UK helps them to understand what's happening when you start a container.

Perhaps the best way to get to grips with containers and container images is to create and work with them. In this lesson, we'll work through some examples that show you how to do the following:

- Create a container image to bundle your Go code with static file dependencies
- Feed environment variables into containers, and open ports so that you can get requests into them

Like many things in software engineering, there are several different approaches you can take to achieve the same thing. In this lesson, we'll look at some options for exposing ports and passing in environment variables to your code. Armed with knowledge of how these different approaches work, you should be in a good position to think about what makes most sense for your Go (and other) projects.

Bundling Go Code into a Container Image

If you're going to work along with the example code, you should change directory to *$GOPATH/src/github.com/lizrice/hello-container-world/Example1*.

Let's begin with very simple web server built in Go that uses static template files. This *main.go* file does all the work:

```
package main

import (
  "html/template"
  "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
  t, err := template.ParseFiles("templates/page.html")
  if err != nil {
    panic(err)
  }
  t.Execute(w, nil)
}

func main() {
  http.HandleFunc("/", handler)
  http.ListenAndServe(":8080", nil)
}
```

This loads the page content from a template file called *page.html*, that the server code expects to find in a directory called templates. Here's a very simple example of a template:

```
<h1>Hello!</h1>

<p>I'd tell you a joke about integers, but there would be no
point</p>
```

You can run the following to check that it works:

```
$ go run main.go
```

In your browser, navigate to 0.0.0.0:8080 (or localhost:8080), and you should see the web page.

Now let's build it into a standalone executable:

```
$ go build -o hello .
```

Running `ls -ltr` shows us what has been built:

```
$ ls -ltr
total 14816
-rw-r--r--@ 1 liz  staff       87  7 Feb 13:58 page.html
-rw-r--r--@ 1 liz  staff      256  7 Feb 14:05 main.go
-rwxr-xr-x  1 liz  staff  7574044  7 Feb 14:16 hello
```

You can run the executable, reload the web page, and all should be exactly as before:

```
$ ./hello
```

Let's try copying that file somewhere else and running it. In this example, I'm copying it to my home directory, moving into that home directory, and then running the code:

```
$ cp hello ~
$ cd ~
$ ./hello
```

Things look fine until you reload the web page in the browser. At that point, the web server goes to look for *templates/page.html*. In the absence of any other path, the only place it looks for the templates directory is the directory that you ran the code in—in this case, the home directory. But there's nothing called *templates/page.html* in that directory! You'll see an error like this:

```
http: panic serving [::1]:62402: open templates/page.html: no
such file or directory
```

The executable binary that we've built refers to an external static file. If you want to deploy this executable (for example, on another machine or in a VM in the cloud) you'll need to ensure that you copy the static file along with the executable.

That's trivial if you only have one template file, but a real web site will likely have dozens, hundreds, maybe even thousands of static files. It would be nice to bundle everything up into one place so that they can be moved around together.

There are a couple of options for this. One approach is to use go-bindata to convert the static content into native Go code that you can then compile into the executable. Another option—and it's the one we'll discuss here—is to include the files inside a container image.

Building a Container Image

Here is the Dockerfile that describes what we want inside our container:

```
FROM scratch

EXPOSE 8080

COPY hello /
COPY templates templates

CMD ["/hello"]
```

Let's look at this line by line.

FROM scratch

The Dockerfile starts with a `FROM` line that gives a starting point for the image we're building. In this example, our go binary executable and its accompanying template are all we need, and we don't have any dependencies. That means we can start from scratch, literally. "FROM scratch" means there's nothing else in our container aside from what we put in with the rest of the Dockerfile.

EXPOSE 8080

The `EXPOSE` directive informs Docker about any ports we want to be able to access. If we don't open ports, we won't be able to access them from outside the container. Because the web server listens on port 8080, we need to be able to send requests to that port.

We could omit this line and instruct Docker what ports to open at the point we run the container—we'll see that approach in the next section—but for now, because we have port 8080 hardcoded into the web server code, we always want port 8080, and it's perfectly reasonable to define it in the Dockerfile so that it's built in to the container image.

COPY hello /

The `COPY` directive copies files or directories from the build context (the directory in which you run the build) into the container. Here we're copying the hello executable into the root directory of the container.

COPY templates templates

Here we copy the contents of the templates directory into the container.

CMD ["/hello"]

This line directs the container as to which command to execute when the container is run. Why the square brackets? If you omit them, this is the shell form of CMD, and Docker will try to execute the command in a shell. More specifically, it will run `/bin/sh -c <command>`. But that won't work for us, because we began from scratch we don't even have /bin/sh inside the container!

A Linux Executable

Whatever operating system you're using, the code that runs inside a container needs to be a Linux binary. Fortunately, this is really simple to obtain, thanks to the cross-compilation support in Go:

```
$ GOOS=linux go build -o hello .
```

If you're on Mac OS X or Windows, you won't be able to run the binary this produces as-is on your machine, but you can run it inside a container.

Building the Container Image

Now, we need to run a command to build the container image:

```
$ docker image build -t hello:1 .
```

The `-t` flag determines what we're going to call this container—for now, I have gone with `hello:1`, meaning that I'm giving it the name "hello" and a tag "1" to indicate which example it's from. The dot instructs Docker to build in the context of the current directory. By default, Docker looks for a Dockerfile called Dockerfile.

As programmers in a compiled language, we're used the idea that a build would generate some sort of executable file, but a Docker image build doesn't create anything that you can readily see in the filesystem. (It does write information about the image to disk, but as a user, you're not expected to know or care about that.) Instead of looking in the file system for your image, you use a `docker` command to list the images.

```
$ docker image ls
REPOSITORY      TAG                 IMAGE ID
CREATED             SIZE
hello           1                   6126f6757608        19
seconds ago      7.57 MB
```

You should see "hello" (if that's what you called it) in the list of images. One thing that might be of interest is the size: the container image is only a little bigger than its contents (which are dominated by 7.2 MB in the executable):

```
$ ls -lh hello
-rwxr-xr-x 1 liz staff 7.2M 9 Feb 11:49 hello
```

This container image is "deployable," meaning that it can be run anywhere that there's a docker engine running. You could store it in

a container registry, and from there it could be pulled onto any machines that will run it. But for the moment we want to run it locally and check that it contains the web server, including its static file directory.

Running the Container

You can run the container as follows:

```
$ docker container run -P hello:1
```

The `-P` flag directs Docker to expose the ports that were specified in the Dockerfile.

Note that in contrast to running executable files directly (where you need to specify the location, or for the file to be in your path), you can run this from any directory.

Open another terminal window to have a look at what containers are running:

```
$ docker ps
CONTAINER ID        IMAGE               COM-
MAND                  CREATED             STATUS
PORTS                     NAMES
d5963ca57417        hello:1             "/
hello"                4 seconds ago       Up 2 seconds
0.0.0.0:32779->8080/tcp   reverent_archimedes
```

We can see that a container has been started and given the random name "reverent_archimedes" (of course, you'll see something different). It's executing "/hello" (as specified by the CMD line in the Dockerfile).

The `PORTS` column shows us how ports exposed on the container can be accessed from the host. In the example, Docker has assigned port 32779; yours will likely vary.

You can check this is the case by browsing to 0.0.0.0:<*port*> (e.g., 0.0.0.0:32779), where you should see the web server running.

Digging Deeper: What's Inside the Container?

I said earlier that there's no */bin/sh* within the container, but you might not want to accept that without further investigation! Let's see exactly what has been built from scratch.

Now, many container images are built from a base image that includes Linux such as Ubuntu, CentOS, or Alpine. If you have one of those you can run a command like `docker exec -it <container> /bin/bash` to get a shell, inside which you could run your favorite commands like `ls` to have a good look around.

But in this case, we don't have */bin/bash* (or even */bin/sh*) within our container, and that `docker exec` command results in an error:

```
$ docker container exec -it reverent_archimedes /bin/sh
rpc error: code = 2 desc = oci runtime error: exec failed: con-
tainer_linux.go:247: starting container process caused "exec:
\"/bin/sh\": stat /bin/sh: no such file or directory"
```

We can't run a command within the container to look at its file system, but what we can do is take a snapshot of the filesystem by using the `docker export` command. This creates a tar archive of the container's file system, which we can then examine by using `tar -xvf`:

```
$ docker container export -o output reverent_archimedes
$ tar -xvf output
x .dockerenv
x dev/
x dev/console
x dev/pts/
x dev/shm/
x etc/
x etc/hostname
x etc/hosts
x etc/mtab
x etc/resolv.conf
x hello
x proc/
x sys/
x templates/
x templates/page.html
```

The first thing to notice is that the files we copied into the container are just as you'd expect: the hello executable is in the root directory, and the templates directory (along with its contents) is there, too.

There is a *.dockerenv* file—you can use `cat` to verify that it's empty.

And, there are four more directories inside the root directory. You probably don't need to know the details but very briefly they are pseudo file systems for Linux:

`dev`
: This holds device files—we can see interfaces to the system console, pseudo-terminals (`pts`) and shared memory (`shm`).

`etc`
: This holds system config files used by the container.

`proc`
: This holds information about processes running in the container.

`sys`
: This holds nonprocess status information.

Even if you're running on a Mac or Windows, the container image is Linux-y.

Why Build from Scratch?

You'll see a very large number of containers that are built on top of a Linux base image. Their Dockerfiles begin with a line that specifies the base image, for example `FROM alpine` or `FROM ubuntu`. But as a Go programmer, you might well be better off building from scratch for a couple of reasons.

Smaller images

Many programming languages, particularly scripted ones like Ruby or Python, rely on some components—not least of which the actual executable that will run your code. You would typically need to install these components on top of a Linux distribution. There are official Docker images that developers can start with these components already baked in, and some of them are pretty complicated. For example (at least as of this writing), the latest version of the Python official image uses buildpack-deps, which in turn is built on the debian:jessie image.

All this code adds up to a pretty sizeable image. The Python image is nearly 700 MB, and the base Debian image alone is 123 MB. This is a lot more than the image we built from scratch, which was less than 7 MB.

NOTE

If you decide that you really do want Linux functionality within your container, there is a pared-down distribution called Alpine that is a much more reasonable 4 MB.

```
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
debian              jessie              978d85d02b87        10
hours ago        123 MB
python              latest              3984f3aafbc9        2
weeks ago         690 MB
alpine              latest              88e169ea8f46        2
months ago        3.98 MB
```

As a Go programmer, you don't (as a rule) need any of this Linux code to run your binary. So, there's no real need to build it in to your image.

Smaller attack surface

The less code there is within your container, the less likely it is to include a vulnerability. Thus, it's good practice to leave out anything you don't need. As an illustration, remember the ShellShock issue that affects bash? If your container doesn't have bash in the first place, it couldn't possibly be affected. The host machine might be, but the point is that you wouldn't need to worry about applying patches to container images as well as the host if those containers don't have the affected code.

Are there any downsides? Well, as we've already seen, you can't simply run a shell like `/bin/bash` within your container, which could be seen as a lack of convenience, or as a security benefit, depending on your perspective. In a future lesson, we'll look at a method for working around this, so you can use scratch-based container images and still get the benefits of being able to run your favorite commands.

Cleaning Up

You can stop the running container by pressing Ctrl-C but that doesn't get rid of it, it merely stops it. You can list all of the containers (including those that aren't running), as follows:

```
$ docker container ls -a
CONTAINER ID        IMAGE               COM-
MAND                  CREATED             STA-
TUS                         PORTS                    NAMES
```

```
d5963ca57417        hello:1             "/
hello"                  44 seconds ago       Exited (2) 42
seconds ago                                  reverent_archimedes
```

Note that this is a container, not the container image. When you start a container, it gets its own copy of the filesystem image that was defined in the container image. You can have many containers on your system (either running or stopped) that were all started from the same container image.

To clean up the container, do this:

```
$ docker container rm reverent_archimedes
```

You'll see the container has gone if you rerun the `ls -a` command.

Cleaning up containers as you go along like this can become pretty tedious, but there are a couple of alternatives:

- Add the `--rm` flag when you run the container (e.g., `docker run --rm -P hello:1`), and it will be removed when it is stopped.
- Whenever you want to tidy up old, stopped containers you can simply run `docker container prune`.

Summary

In this section, we've seen a basic example of building and running Go code within a container. Here are the important takeaways:

- There are two steps to the build: compiling the Go code, and then building the container image.
- The container image has everything we need to execute the code (and nothing else, because we started from scratch).

We talked about the pros and cons of building container images from scratch (the efficient, slimline approach) or on top of a Linux distribution (which gives you the option to run Linux commands within the container, which could be convenient or a security risk, depending on your point of view).

You have also tried out several useful Docker commands.

Image Commands

`docker image build`
: Builds a container image

`docker image ls`
: Lists container images found on your local machine

Container commands

`docker container run`
: Creates a container from a container image, and runs code inside the container

`docker container exec`
: Connects to a container and runs another command

`docker container export`
: Exports a snapshot of the container file system as a tar archive

`docker container ls`
: Lists running containers

`docker container prune`
: Removes stopped containers

We have also seen how, when you start a container, Docker can make ports available from the container to the host. Those ports are not accessible by default; you must ask Docker to expose those ports so that you can access them.

Further, the port number on the host can be completely different from the port number within the container—or put another way, the code within the container is using some port number *X*, but if you want to access it from the host, you need to address the code using port *Y*; Docker does the mapping from port X to port Y.

In the next section, you'll learn more about how you can control these port mappings, and also how you can supply configuration information to your containerized code through the use of environment variables.

Environment Variables and Port Mappings

Let's modify the web server code so that it uses an environment variable to determine the port on which it should listen. Change the `main()` function in *main.go* to this (or use the code in the *Example2* directory):

```
func main() {
  port := os.Getenv("WEB_SERVER_PORT")
  if port == "" {
    port = ":8080"
  }
fmt.Printf("Serving on port %s\n", port)
  http.HandleFunc("/", handler)
  http.ListenAndServe(port, nil)
}
```

This defaults to 8080, but you can change the value by setting an environment variable. (For brevity I'm holding the port value as a string including the preceding colon that `http.ListenAndServe` expects. In production code, you'd probably want to allow the environment variable to be only the port number, without the colon.)

You can try this out locally on your machine (from inside the *Example2* directory):

```
$ export WEB_SERVER_PORT=":8081"
$ go build -o hello .
$ ./hello
Serving on port :8081
```

Now, you should find the service by browsing to 0.0.0.0:8081.

When we run the same code within a container, we need a way to define the environment variable so that the code knows what port to use.

Let's explore some options for passing the environment variable into the code when it's running in a container.

Building env vars into the container image

The first option is to define the environment variable within the Dockerfile. We'll do this by adding the following line (you'll see this in the Example2 Dockerfile):

```
ENV WEB_SERVER_PORT :8081
```

We can then refer to that variable to define what port we'll expose:

```
EXPOSE $WEB_SERVER_PORT
```

Try that out by building and running the container as follows:

```
$ GOOS=linux go build -o hello .
$ docker build -t hello:2 .
$ docker run -P --rm hello:2
Serving on port :8081
```

The code tells us it's listening on port 8081, but that's only true from its perspective within the container. As before, this will have been mapped to a different port on the host, which we can see by looking at the running containers:

```
$ docker container ls
CONTAINER ID        IMAGE               COM-
MAND                  CREATED              STATUS
PORTS                     NAMES
3e8d5501f7a3        hello:2             "/
hello"                About a minute ago   Up About a
minute   0.0.0.0:32789->8081/tcp   nervous_khorana
```

We can see the website up and running by browsing to 0.0.0.0:32789 (remember to use the port that you see in the output of `docker container ls`).

When you stop the container (Ctrl-C will do it) the `--rm` flag instructs Docker to remove the container altogether (we mentioned this briefly in the previous section).

Overriding the env var

You can use the `-e` flag to override environment variables defined in the Dockerfile, but beware: this might not be sufficient!

Let's override `WEB_SERVER_PORT` at the point we run the container:

```
$ docker run --rm -P -e WEB_SERVER_PORT=:8082 hello:2
Serving on port :8082
```

This looks good. We can see that the executable has picked up the definition of `WEB_SERVER_PORT` from the `-e` flag, but take a look at the running container and its port assignments:

```
$ docker container ls
CONTAINER ID        IMAGE               COM-
MAND                  CREATED             STATUS
PORTS                     NAMES
827838bbbb5c        hello:2             "/
hello"                12 seconds ago      Up 10 seconds
0.0.0.0:32792->8081/tcp   happy_poincare
```

The problem is that the EXPOSE command was built in to the container image with a port definition of 8081. The Go executable inside the container is happily serving on port 8082, but that's not accessible from the host! You can try browsing to 0.0.0.0:32792, but there will be nothing to see.

Fortunately, we can tell Docker what port to expose when we run the container, as follows:

```
$ docker run --rm -p 8082 -e WEB_SERVER_PORT=:8082 hello:2
Serving on port :8082
```

This time, the lowercase `p` flag tells Docker which port we want opened up, and this matches the port number we've asked the executable to serve on (via the environment variable):

```
$ docker container ls
CONTAINER ID        IMAGE               COM-
MAND                  CREATED              STATUS
PORTS                                NAMES
7b1e68f267fa        hello:2             "/
hello"                About a minute ago   Up About a
minute   8081/tcp, 0.0.0.0:32793->8082/tcp   nervous_ramanujan
```

The host port 32793 is now mapped to port 8082, and we can browse our web server on that port successfully.

Specifying the Port Mapping

It's getting a bit tedious having to look up the host port that Docker assigns when we run a container. If you prefer, you can specify the container port–to–host port mapping. In the following example, `-p <host port>:<container port>`, we instruct Docker as to which web service to make available, in this case, host port 8000.

```
$ docker run --rm -p 8000:8082 -e WEB_SERVER_PORT=:8082 hello:2
Serving on port :8082
```

You can browse to 0.0.0.0:8000 and find the web page there.

Back to the Dockerfile

If we're going to specify the port mapping and environment variables at runtime, we might as well take the `EXPOSE` and `ENV` directives out of our Dockerfile. It's good practice to keep your Dockerfiles as short as possible because each directive adds a file system layer and increases the build time.

You can prove that this works by building and running the container with the provided, shorter Dockerfile2:

```
$ docker build -f Dockerfile2 -t hello:2 .
$ docker run --rm -p 8000:8082 -e WEB_SERVER_PORT=:8082 hello:2
Serving on port :8082
```

As before, the server is available at 0.0.0.0:8000 (or localhost:8000).

Summary

You can use environment variables to pass configuration information to your containerized code (in true 12-factor app form). You also need to expose container ports so that you can make requests of your code.

We've seen that you have choices when it comes to defining these environment variables and exposed ports: you can build them in to the container image by defining them in the Dockerfile, or you can set them at runtime on the command line.

If you define these things in the Dockerfile, they are built into the container image, and you don't need to remember to add them at runtime. This is a good approach for things that are hardcoded; for example, in the previous section when we had a hardcoded port in the Go code, there was no reason not to define the exposed port at build time by defining it in the Dockerfile. Defining the port with an environment variable in the Dockerfile, as we did at the beginning of this section, is also a perfectly sensible approach when you have no reason to override it at runtime.

Another example for which it really makes sense to define at build time through the Dockerfile would be an environment variable that instructs the code how to behave on particular hardware platforms. This isn't something that you'll want to change at runtime.

But when ports or environment variables might change according to the context that they're running in, you might prefer to define them when you run the container. As with many things in containers (and in software engineering in general!) there are several ways to skin a cat, and you must choose which is best for your particular application. If you've worked through the examples, you should have a helpful grip on both approaches.

And I'm afraid there is yet another way to define values like the ports and environment variable post-build: Docker Compose. This

really comes into its own when you're using multiple containers that interact with one another, which we'll cover in an upcoming lesson.

Recap

In this lesson, you've seen the following:

- How to build Go code into a container
- The basics of what to put in a Dockerfile
- How Go programmers can take advantage of building from scratch rather than a Linux base container image
- How to pass environment variables to your code, either at image build time or at runtime
- How to control port mappings between the container and the host on which it's running.

About the Author

Liz Rice, cofounder of Microscaling Systems, has a wealth of software development, architecture management experience in technology sectors including VOD, music, VoIP and the charity sector. When not building startups and writing code, she loves riding bikes in places with better weather than her native London.