How to Build Your Own Custom Buildpacks

Guide

www.suse.com

Application Delivery

With SUSE Cloud Application Platform, it's possible to build your own custom buildpacks. These are useful for a variety of reasons: locking down versions of a framework, adding in custom logic, etc.. This process is reasonably straightforward.

In this example buildpack, I will install a version of node.js and incorporate a simple function wrapper. For the purposes of this example, this will be done inside a single buildpack. It will also be done purely in Bash to provide more clarity on how everything works together. (Please forgive my poorly written bash!)

In a future guide, I'll tackle how to split this logic into multiple buildpacks.

How a Buildpack Works

We need to know how each of the scripts are used together so that we can build one ourselves. This is not overly complicated, but can be a bit confusing until you try it out.

Buildpack Scripts

There are four scripts that a Buildpack needs to implement:

Detect

The buildpack decides whether it can/will be used. Its exit value is either 0 for true or 1 for false. This step is skipped if the application manifest specifies the buildpack(s) to be used.

Supply

Sets up any needed dependencies. This is run for each buildpack in the specified list or for the first buildpack that returns 0 for its detect script.

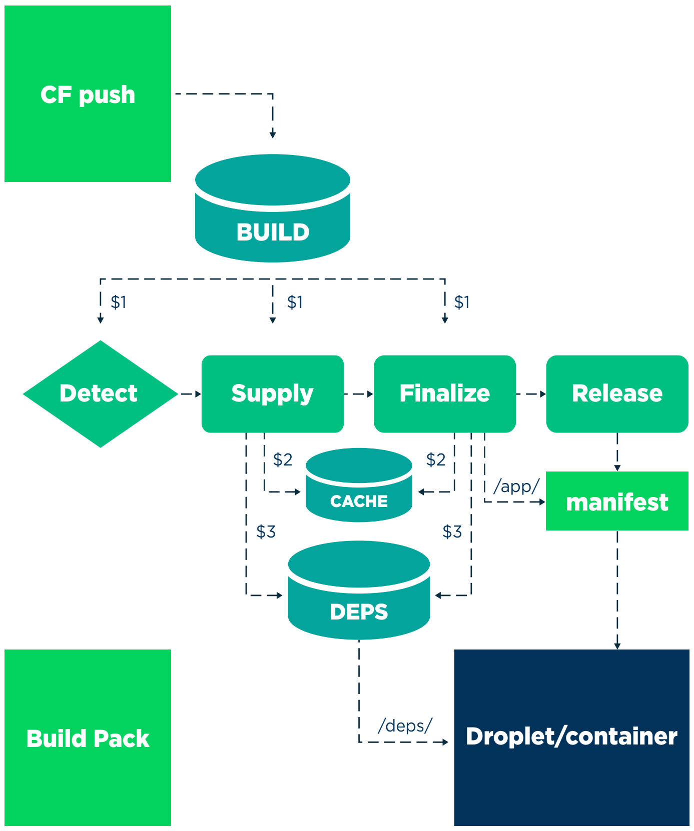
Finalize

Assembles, builds or compiles the app and moves everything into the correct folder structure. (This only runs for the last buildpack in the chain.)

Release

This script specifies how to run the built application. This is done by writing some yaml to the script's standard out. (This is also only run for the last buildpack.)

Because there are a lot of moving parts, which can get confusing quickly, here's a diagram that shows the relationships between the folders and the scripts:



Directories

There are a few directories that we need to know about while writing these scripts.

Buildpack Source

The buildpack source can be found using the line:

```
export BUILDPACK_DIR=`dirname $(readlink -f
${BASH_SOURCE%/*})`
```

This is useful if you are adding extra files packed into the buildpack.

Build Directory

This contains the application uploaded by cf push.
The Build Directory is passed in as the first argument to both the Supply and Finalize scripts:

```
  export BUILD=$1
```

Cache Directory

The cache is used to hold files between stages of the build pipeline, as well as between builds:

```
  export CACHE=$2
```

We will use this as a staging area.

Dependencies Directory

This is actually a set of directories, one for each buildpack in the chain. The current one can be found with this, where $3 is the general directory and $4 is the index of the buildpack:

```
  export DEPS=$3/$4
```

Droplet Structure

The output of the buildpack staging process will be a droplet. This is the file structure that will then get built into the diego cell (or container with Eirini enabled!).

The built directory gets mounted to /home/vcap/app/ (and symlinked to /app/).

The dependencies directories get mounted to /home/vcap/deps/<id>/.

The manifest written by the release script gets loaded into the config and will be run by the script /lifecycle/launch. You don't need to worry about this script. It just kicks off and manages the application that gets built.

Manifest

A buildpack itself has a manifest file that can control a lot of stuff about how it gets used. There are a lot of options that can be in this yaml, which I'll likely write about in the future.

Build Your Own Buildpack

Now that we have some understanding of how a buildpack works, let's build a simple, custom one. In this guide, I'll show how to create a buildpack that builds a single javascript function into a routable service.

The finished buildpack can be found at: https://github.com/agracey/custombuildpack/

Base Directory Structure

We need to create a directory with the following files:

```
  ./bin/detect
  ./bin/supply
  ./bin/finalize
  ./bin/release
  ./manifest.yml
```

We will also add a file app.js to act as the wrapper. My take on a function wrapper can be found here.

./bin/detect

Decide whether the buildpack should even run. The application directory is passed in as $1.

We will require the user to have a file containing the function, as well as a package.json with any additional dependencies.

This can be achieved in bash with:

```
    if [ -f $1/package.json && -f $1/function.js ];
    then
      echo "node function`cat $BP/VERSION`" && exit 0
    else
      echo "no" && exit 1
    fi
```

The way this works is that if it sees both files, it prints the version and returns 0. If it doesn't see one or the other, it prints "no" and returns a 1.

./bin/supply

Pulls in any needed dependencies.

We need to pull down and unpack the node.js engine and npm. This comes as a single tarball. Releases can be found at: https://nodejs.org/download/release/

```
curl https://nodejs.org/download/release/v12.5.0/
 node-v12.5.0-linux-x64.tar.gz > $CACHE_DIR/node.tgz
tar -xzf $CACHE_DIR/node.tgz -C $DEPS_DIR/
```

Let's download into the cache and unpack into the directory. (I'm not going to complicate things by optimizing at the moment, so let's ignore that the cache could be a cache for now.)

./bin/finalize

Builds the final project into the right structure.

In this step, we need to pull together the files from the uploaded application, dependencies, and any files needed from the build-pack itself.

First let's pull all the files together. We can source the function wrapper from the buildpack directory and the rest of the needed files from the cache directory:

```
cp $BUILDPACK_DIR/app.js .
cp $CACHE_DIR/function.js .
cp $CACHE_DIR/package.json .
cp $CACHE_DIR/package-lock.json .
```

Next, we need to run npm install to pull down any dependencies. The trick with this is that npm relies on the node being in the $PATH, so we can't just call it directly with no prep. Luckily, this is easy to remedy:

```
export PATH=$DEPS_DIR/node/bin:$PATH
```

Now we can run npm install to get our

```
npm install
```

With that done, everything is in place and built. This can obviously be much more complicated if needed, but I tend to prefer the simpler solution.

./bin/release

Instructs the runner how to run what was just built.

This step is a bit weird in that you must write yaml to standard out and it needs to be in the form:

```
default_process_types:
     web: <runner>
```

The working directory that this is running in is /home/vcap/app/. We need to find our executable dependency. Since we copied our node binary to the dependencies folder in the supply script, we can find it in /home/vcap/deps/0/.

So, this gives us a script of:

```
echo "default_process_types:"
echo " web: /home/vcap/deps/0/node/bin/node app.js"
```

manifest.yml

Lastly, we need to write the yaml for the buildpack manifest itself, because some tools will use this.

Due to the simplicity of the sample buildpack we are building here, this can just be a list of files that need to be included:

```
---
include_files:
- bin/detect
- bin/supply
- bin/finalize
- bin/release
- manifest.yml
- app.js
```

Add Your New Buildpack to CloudFoundry

To use your new buildpack, you need to create a zip file containing the directory structure:

```
$ pushd .. && zip -r custombuildpack.zip custom-
 buildpack/ && popd
```

With this zip file, you can upload it to the Cloud Application Platform:

```
$ cf create-buildpack custombuildpack custom-
 buildpack.zip 1
```

You can see which buildpacks are in your system with:

```
$ cf buildpacks
```

Use Your New Buildpack

Now you can build an application using this new buildpack. In a separate folder, you need these files: manifest.yml, function.js, package.json.

An example application using the code can be found at: **https://github.com/agracey/custombuildpack_exampleapp**

The package.json can be generated by following the prompts from: (or just copy the example)

```
$ npm init
```

The function.js can be as simple as:

```
var exports = module.exports = {

    run: (event, context) =>{
    console.log('HELLO CONSOLE!', JSON.
 stringify(event), JSON.stringify(context))
    return "Hello World"
    }
}
```

The manifest needs to give a name to the application and specify which buildpack should be used:

```
applications:
- name: test-with-custom-function
- buildpacks: [ custombuildpacks ]
```

With these files created, run your new app with:

```
$ cf push
```

As part of the output, it should give you the route that is created, which you can now browse to.

Additional Notes

Custom buildpacks can be an incredibly powerful way to simplify your application delivery lifecycle. You can add your own commonly used patterns and frameworks in an easily manageable way.

With this flexibility, you can make SUSE Cloud Application Platform work the way your organization needs it to work. It also allows you to move complexity to a more appropriate place in the process.

As noted above, this guide will likely be followed by exploring even more interesting ways to use this. As a sneak peak, I'm hoping to look at: multi-buildpack scenarios, airgapped buildpacks, and caching in buildpacks for optimization.

Additional contact information and office locations:
www.suse.com

www.suse.com