cpk

Andrea F. Daniele

Sep 06, 2021

CONTENTS:

cpk is a toolkit that standardize the way code in a project is structured and packaged for maximum portability, readability and maintainability.

CHAPTER

ONE

INTRODUCTION

cpk stands for Code Packaging toolKit and is designed to standardize the way code in a project is structured and packaged for maximum portability, readability and maintainability.

cpk is the result of years of experience in the context of cross-user, cross-machine, cross-architecture development and deployment of software modules. Originally created to standardize and simplify code development and deployment in Duckietown, it later became an independent toolkit.

CHAPTER

TWO

FEATURES

cpk organizes code in **projects**. A *cpk* project is a directory containing everything that is needed for the project to be built, packaged, documented and deployed.

The power of *cpk* comes from the technologies it is built on:

- **Python** (for cross-platform availability);
- **Git** (for code versioning);
- **Catkin** (for source code packaging and dependencies management);
- **Docker** (for code packaging and deployment);
- **QEMU** (for cross-platform code building and deployment);
- **SSH** (for fast and secure communication between build and deployment nodes);
- **rsync** (for reliable code synchronization);

The next two sections will jump straight into how to install *cpk*, then build and run a simple *cpk* project. Don't miss them.

CHAPTER

THREE

INSTALLATION

You can install *cpk* through *pip*.

```
$ pip3 install cpk
```

CHAPTER

FOUR

GET STARTED

We will now create, build and run an empty project, we will then take a step back and examine how to populate the project with your own code.

4.1 Create an empty project

Use the following command to create an empty cpk project.

```
$ cpk create ./my_project
```

You will be asked to provide information about your new project. For example,

```
cpk|    INFO : Please, provide information about your new project:
   |
   |        Project Name:           my_project
   |        Project Description:    My best project
   |        Owner Username:         afdaniele
   |        Owner Full Name:        Andrea Daniele
```

4.2 Build the project

Now that our empty project is created, let's build it.

```
$ cd ./my_project
$ cpk build
```

Let it build and you will see a summary of the build that looks like the following,

```
...
================================================================================
Final image name: afdaniele/my_project:latest-amd64
Base image size: 120.25 MB
Final image size: 120.25 MB
Your image added 1.08 KB to the base image.
------------------------
Layers total: 48
 - Built: 48
 - Cached: 0
```

(continues on next page)

(continued from previous page)

```
------------------------
Image launchers:
 - default
------------------------
Time: 5 seconds
Documentation: Skipped
================================================================================
```

This means that the project was built successfully, now let's run it.

4.3 Run the project

```
$ cpk run
```

You will see the following output,

```
...
==> Entrypoint
<== Entrypoint
This is an empty launch script. Update it to launch your application.
```

This means that our project run correctly. Congratulations, you just built and run your first cpk project.

The following sections will teach you how to,

- *Add code to a project*;
- *Use remote machines*;

CHAPTER

FIVE

ADD CODE TO A PROJECT

Code in *cpk* project is organized in packages. *cpk* supports:

- Catkin Packages;
- Python 3 (Module) Packages;

Both *Catkin* and *Python* packages are basically directories with a specific structure. Place your *Catkin* and *Python* packages inside the `packages/` directory in your *cpk* project. This is enough for *cpk* to detect them and configure them for use inside the Docker container.

Let's see how to add packages to our project.

5.1 Launchers

Before we can dive into how to add code to a *cpk* project, let's talk about the concept of **launchers**. A *launcher* is an executable file that you can pick as the entrypoint when your project runs. In other words, the launcher will be the first process that gets executed in the container when we do `cpk run`.

A *cpk* project can have multiple launchers, and they are stored in the `launchers/` directory at the root of our *cpk* project. There is always a *default* launcher inside that directory. The default launcher is a bash script that prints out the string *"This is an empty launch script. Update it to launch your application."* and exits. And that is exactly what you see when you execute the command `cpk run` from inside a newly created project.

Any executable file, or script file beginning with a shebang, is detected by *cpk* as a valid launcher.

Changing the content of the default launcher is usually enough for simple applications, but more complex projects might need multiple launchers, you can create as many as you need inside the launchers directory.

An example of multi-launcher project could be one in which the default launcher runs an application in "Release" mode while a secondary `debug` launcher launches it in "Debug" mode.

Use the argument `-L/--launcher` of `cpk run` to run a non-default launcher.

5.2 Create a Python Package

From the root of a *cpk* project, let's move to the packages directory and create a Python package inside called `my_python_package` that is compliant with the schema of a Python package. If you are not familiar with the Python package schema, you can learn more by reading the official documentation.

```
$ cd ./packages/
$ mkdir ./my_python_package
```

(continues on next page)

(continued from previous page)

```
$ cd ./my_python_package
$ touch __init__.py
```

The snippet above creates the simplest Python package possible, which consists of an empty directory called `my_python_package` containing an empty file called `__init__.py`.

We can now add a Python module to the Python package we just created. Let's create a very simple module called `main.py` inside the directory `my_python_package/` with the following content,

```
if __name__ == "__main__":
    print("Hello from Python")
```

The module above implements the classic Python *"Hello world"* example. Let's build it and run it using *cpk*. We will begin by telling our default launcher that this is new module is the application we want to run. We can do so by updating the content of the file `launchers/default.sh` to the following,

```
#!/bin/bash
python3 -m my_python_package.main
```

We can now build and run our project using the commands,

```
$ cpk build
$ cpk run
```

If everything went well, we should see something like the following,

```
...
Hello from Python
```

This is all we need to know to start packing our *cpk* projects with custom code. As you might have noticed, *cpk* took care of discovering our `my_python_package` package and adding it to the `PYTHONPATH` environment variable.

CHAPTER

SIX

ADD DEPENDENCIES TO A PROJECT

Dependencies are libraries and tools our application relies on at build or run-time. They are usually installed via package managers, like **Aptitude** (`apt` or `apt-get`), the **Python Package Index**'s `pip`, etc. *cpk* supports both `apt` and `pip3` package managers.

6.1 Add `apt` dependency

We can list our dependency packages installable through the `apt` package manager in the file `dependencies-apt.txt` available at the root of our *cpk* project.

cpk allows us to add comments and blank lines in this file, this is useful when we want to group dependencies together and keep track of what each dependency is needed for. For example, a valid `apt` dependencies file is the following,

```
# generic tools (this is a comment)
git

# dependencies for feature A
libA
libB

# dependencies for feature B
libC
```

6.2 Add `pip` dependency

We can list our dependency packages installable through the `pip3` package manager in the file `dependencies-py3.txt` available at the root of our *cpk* project.

Similar to what we can do in `dependencies-apt.txt`, *cpk* allows us to add comments and blank lines in this file.

A valid `pip3` dependencies file is the following,

```
# generic tools (this is a comment)
numpy
scipy

# dependencies for feature A
flask
```

CHAPTER

SEVEN

USE REMOTE MACHINES

The term **machine** in cpk is used to indicate an endpoint, that is reachable over the internet and on which we want to build and run our projects.

It is convenient to let *cpk* handle our machines, so that we don't have to insert passwords or type in long and hard to remember hostnames or IP addresses.

For example, we are working on a new project and we want to be able to test it on a workstation in our office while we are working from home, maybe using a not too powerful laptop. In this case, we would register our workstation as a *cpk* machine and then tell cpk to build and run our project there instead of our laptop.

Machines are managed using the command `cpk machine`.

7.1 Create a Machine

Using the example above, we assume that our workstation is reachable at the IP address *10.0.0.1*. We have two ways of connecting to our workstation, TCP or SSH.

7.1.1 Create an SSH Machine (recommended)

We assume that our workstation is configured to accept SSH connections and that we have an account with username *myuser* on our workstation.

We create a *cpk* machine called *myws* using the command,

```
$ cpk machine create myws myuser@10.0.0.1
```

Connections based on SSH are made using 2048 bit RSA keypairs that *cpk* creates and exchanges with the destination machine. For *cpk* to install the key on the destination, we will be prompted to insert the user password.

Note: The SSH password is only needed to transfer the keys, it is not stored and/or used by cpk for anything else.

Once the keys are transferred, we are ready to use the new machine.

7.1.2 Create a TCP Machine (unsecure, not recommended)

> **Warning:** TCP connections are not encrypted and require that the destination exposes the Docker endpoint to the network. This is very dangerous and only suggested within a private local network.

We create a *cpk* machine called *myws* using the command,

```
$ cpk machine create myws 10.0.0.1
```

7.2 List Machines

We can list the machines stored by *cpk* by running the command,

```
$ cpk machine list
```

A shortcut for the command above is `cpk machine ls`.

7.3 Remove a Machine

We can remove a machine called `myws` using the command,

```
$ cpk machine remove myws
```

A shortcut for the command above is `cpk machine rm`.

7.4 Use a Machine

You can think of *cpk* as having a default machine that gets created when you install it, pointing to your local Docker endpoint.

Once your machine is created, you can redirect all your *cpk* commands towards it, instead of the default (local) machine. In order to do so, pass the argument `-H/--machine <MACHINE>` to your *cpk* commands.

Note:

The value of `<MACHINE>` can be:

- The name of a machine you previously created (e.g., `myws`);
- A (resolvable) hostname of a machine exposing the TCP socket for Docker;
- An IP address of a machine exposing the TCP socket for Docker;

For example, we can show information about the endpoint a machine points to with the command,

```
$ cpk endpoint info -H myws
```

This will show you information about the Docker endpoint the machine `myws` is pointing to, for example,

```
cpk|    INFO : CPK - Code Packaging toolKit - v0.0.4
cpk|    INFO : Retrieving info about Docker endpoint...
   |
   | Docker Endpoint:
   |   Machine: myws
   |   Hostname: myws
   |   Operating System: Ubuntu 20.04.2 LTS
   |   Kernel Version: 5.4.0-74-generic
   |   OSType: linux
   |   Architecture: x86_64
   |   Total Memory: 251.64 GB
   |   CPUs: 64
   |
```

You can build and run your *cpk* projects against a machine, with `cpk build -H myws` and `cpk run -H myws`.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search