palm-cli

Release 2.5.3

Palmetto Data Team

May 16, 2023

CONTENTS

Palm — the extensible CLI at your fingertips!

Check Out Some Example Use Cases

Understanding The Palm Viewpoint: Why do my CLIs need a CLI?

CHAPTER

ONE

WHAT IS PALM?

- A highly configurable, extensible, and flexible CLI framework for data professionals.
- A layer of abstraction across your stack, to help with context shifting.
- A simple mechanism for sharing abstraction and automation across your engineering teams.
- A tool for building and managing your own CLI commands.
- Built with python and click

1.1 Contents

1.1.1 Usage

Palm CLI is a command line tool, once installed you can use palm by running `palm` from your terminal. If no command is provided, `palm` will render the help text for commands in the current project

```
> palm
Usage: palm [OPTIONS] COMMAND [ARGS]...

Palm command line interface.

Options:
--version  Show the version and exit.
--help     Show this message and exit.

Commands:
build     Rebuilds the image for the current working directory
docs      Generates internal readthedocs for palm and serves them
plugin    Palm plugin utilities
scaffold  Scaffold new palm commands
update    This updates the current version of palm.
```

System Requirements

Palm is designed to be OS agnostic and should work on Windows, Mac OS X, and common Linux distributions.

Palm requires the following software to be installed and running on your device:

1. Docker You can check to see if you already have it with `docker --version`
2. Python3 You can check to see if you already have it with `python3 --version`

Installation

Install Palm CLI via pip:

`pip install palm`

For development installations, you may also install palm from source by cloning the codebase and running `python3 -m pip install .`

To verify that the installation was successful, run `palm --version`.

note for mac users: if you get this warning:

```
WARNING: The script palm is installed in '/Users/yourname/Library/Python/3.8/bin' which␣
↪is not on PATH.
Consider adding this directory to PATH or, if you prefer to suppress this warning, use --
↪no-warn-script-location.
```

you will need to add `'/Users/yourname/Library/Python/3.8/bin'` to your path for `palm` to work. You can do that with one of these commands (depending on your shell of choice):

- zsh: `echo "\nexport PATH=$PATH:/Users/yourname/Library/Python/3.8/bin\n" >> ~/.zprofile`
- bash: `echo "export PATH=$PATH:/Users/yourname/Library/Python/3.8/bin" >> ~/.bashrc`
- fsh: `echo "setenv PATH $PATH:/Users/yourname/Library/Python/3.8/bin" >> ~/.fshrc`

Configuration

To configure your project to use Palm, run `palm init` from the root directory of your project.

This will walk you through setting up Palm for your project and create a `/.palm` directory containing a `config.yaml` - this is where you can make changes to your project's Palm configuration.

Configuration Options:

- image_name: (str) the name of the docker image used to run your project
- plugins: (list) a list of plugins used by your project, plugins must be installed!
- protected_branches: (list) a list of github branches that palm will not run against

Global palm configuration

As of palm v2.2.0 palm also supports a global configuration file. This file is automatically created at `~/.palm/config.yaml` and contains the following options:

- plugins: (list) a list of plugins used globally, plugins must be installed!
- excluded_commands: (list) a list of palm commands that you do not want to use.

Shell Completion

To enable autocomplete for palm commands, add one of the following shell-specific lines to your shell's profile. Once added, either source your profile or start a new shell session.

zsh: `eval "$(_PALM_COMPLETE=zsh_source palm)"`

bash: `eval "$(_PALM_COMPLETE=bash_source palm)"`

fish: `eval "$(_PALM_COMPLETE=fish_source palm)"`

Adding shell completion to your commands

If you'd like to add/improve shell completion for your own commands, check out the click documentation for shell-completion.

1.1.2 High-level Features

Run in docker

Palm and docker are like peas in a pod. Docker containers allow us to run our commands in a sandboxed environment that is isolated from the rest of the system and is a close reflection of our production and CI environments. Docker ensures that everyone on your team is using a consistent OS and set of dependencies. We're not here to sell you on Docker, and we don't expect you to be a Docker pro, but you will need Docker in order to work with Palm.

Local commands

Palm allows you to define commands within each of your projects, and then share them across your team. Once you have set up your project to use Palm, you can create new commands in the `.palm` directory, add these to version control and they will be available to everyone on your team when they run `palm`.

See the *Commands* section for more information.

Code generation

Palm includes code generation functionality, powered by jinja. Code generation allows you to automate repetitive boiler plating tasks, and keep your codebase consistent. Code generation is driven by the `environment.generate()` function.

See the *Code generation* section for more information.

Plugins

Palm is extensible. You can install plugins that extend the functionality of Palm. Adding commands for specific frameworks or project types is a common use case. Once installed, plugins must be configured for use with your project.

See the *Plugins* section for more information.

1.1.3 Commands

Commands are python scripts that are executed by Palm CLI. They allow you to run complex tasks in a repeatable way, with a simple interface, and ensure everyone on your team is working with the same tools.

Palm commands import the click CLI library, which is a core dependency of palm. Familiarity with the click library is recommended for developing your own commands, many examples can be found in the palm-cli repository.

Where do commands come from?

- Palm ships with some commands out-of-the-box that are always available. These are called "core" commands. For example the `palm build` command is a core command.
- Palm plugins provide commands for working with specific tools. For example, the `palm-dbt` plugin provides commands for working with dbt projects.
- You can create your own commands within your projects, these are called "repo" commands, as they exist only within the project's repository.

Overriding commands

With Palm, it is possible to override commands from other plugins. This is done based on the order of the plugins in the `.palm/config.yaml` file, and the naming of project plugins.

For example, if the `palm-dbt` plugin defined a command named `build` it would override the core `palm build` command. This is because the commands from the `palm-dbt` plugin are added after the core commands. If you installed a second (ficticious) plugin called `palm-builder` which also defined a `build` command, the order of overriding would be determined by the order of plugins in your project's `.palm/config.yaml` file. Finally, if you define a `build` command in your project, as a repo command, it would override all other definitions of the `build` command.

Excluding commands from your project

Sometimes, you may want to exclude commands from your project. This is done by adding the `excluded_commands` configuration to your project's `.palm/config.yaml`.

Example:

Command Groups

Command groups are a useful structure for grouping commands together. For example, if you have multiple commands relating to a single tool, you can group them together within a command group. This allows you to easily see all the commands for a tool when you run `palm <tool name> --help`.

Command groups are a construct provided by click - for more information, see the click documentation on commands and groups

Writing your own commands

To simplify the process of writing your own commands and command groups, Palm ships with some scaffolding commands. These commands will generate boilerplate code for you, allowing you to focus on writing your command's functionality.

- To scaffold a single command you can run `palm scaffold command --name <command-name>`
- To scaffold a command group you can run `palm scaffold group --group <group-name> --command <command-name>`

Once you have scaffolded your command or command group. You can edit the generated file in your .palm directory, add the functionality you need and then run the command immediately with `palm <command-name>`.

Conventions

- Command files are always named `cmd_{name}.py`
- A command _must_ expose a `cli` function. This function is called when the command is executed.

Command Syntax

- The `cli` function should be decorated with either the `@click.command` decorator or the `@click.group` decorator.
- The `cli` function can optionally be decorated with `@click.pass_obj` and accept an `environment` argument, which is a click context object. The `environment` is a useful Environment provided by palm that enables you to perform complex operations, like running in docker containers, generating code, etc.
- For commands within a command group, each command _must_ be decorated with the `@cli.command` decorator. Note that this is different from the `@click.command` decorator, as the command belongs to the `@click.group()` which is always the `cli` function.

Common patterns and important notes

Run in docker:

The global `run_in_docker` function is used to execute a command in the docker container for the current project. This is used in many palm commands. This function is provided via the palm context. If you want to use `run_in_docker` in your own command, ensure you use the `@click.pass_obj` decorator for your command, then use `environment.run_in_docker(command)`.

Run on Host:

Palm provides a simple interface for running shell commands directly on your machine via the context (similarly to how `run_in_docker` is accessed, via `environment`). We highly recommend using `run_on_host` over rolling your own subprocess commands.

> **Warning: Why Not Use subprocess?**
>
> The prime directive of palm is to give all your developers an identical interface and experience, regardless of environment. Different versions of python running on different operating systems can behave differently when calling `subprocess`; palm normalizes this behavior in `environment.run_on_host`.

Importing code:

When writing "repo" commands in your project, you will not be able to use conventional relative imports in your commands, as the command is executed in the context of palm. If you need to share logic between commands, or import code from your project, you must do this with the `environment.import_module` function. This function is provided via the palm context and uses importlib to ensure your shared code is imported from the correct location at run time.

Examples:

Maybe you want a command that kicks off a slow-building container as a background process, but you want to see it complete before moving it back. That could look something like this:

```
## ./palm/cmd_slow_starter.py
...
@click.command('slow_starter')
@click.pass_obj
def cli(environment):
    """Starts the container as daemon, watches the logs, then exits"""
    environment.run_on_host("docker compose run -d super_slow_starting_django_app",
                            check=True)

    ## this is where we watch, pseudo-blocking
    building_logs = str()
    while "Starting local webserver via runserver on port 8080..." \
      not in building_logs:
        logs, _, _ = environment.run_on_host("docker compose logs static_app")
          if logs != building_logs:
              building_logs = logs
              click.echo(logs)
    click.secho("Super-slow app is _finally_ ready!", fg="green")
```

1.1.4 Code generation

Palm includes a set of code generation commands that allow you to generate code for your project, automating repetitive boiler plating tasks, and making your codebase more consistent.

Code generation in palm is powered by Jinja2 and PyYAML.

Basics

To use Palm code generation you will need:

1. A directory of templates. We recommended you make a subdirectory within your project's .palm directory.
2. A YAML configuration file called `template-config.yaml` - this is a configuration file that describes how to generate code.
3. A palm command which is decorated with the `@click.pass_obj` decorator and calls `environment.generate(template_path, output_path, replacements)`.

For full documentation of the generator see `palm/code_generator.py` in the Palm CLI source code.

Template config

The `template-config.yaml` file is a fundamental piece of code generation with palm. It describes the directory structure we want to create, and where each of our templates will generate code.

The config file works on 2 top-level objects: 1. `directories` - a list of directories to create. 2. `templates` - a dict of templates to use, and where to use them. The key is the name of the template, and the value is the path at which we want to create a file from it.

Each line of the template-config is parsed by jinja, which enables you to use replacements in your config file.

Example template config

```
directories:
  - "{{model_name}}"
files:
  - base_model.sql: "{{model_name}}/{{model_name}}.sql"
  - base_model.yml: "{{model_name}}/{{model_name}}.yml"
```

The `replacements` dict allows this template-config to be used to create a directory containing an appropriately named sql file and a yaml file.

Gotchas

Generating code is awesome, but there are gotchas to be aware of.

1. Generating code that contains jinja is a pain, all jinja expressions must be provided as replacements in the template, to prevent jinja from trying to evaluate them during code generation.

New Projects with Cookiecutter

Palm includes a *palm new* command which uses cookiecutter to generate new projects. This is a great way to get started on a new project, and is the recommended way to start a new project.

Default cookiecutter templates

Palm allows you to configure a set of default cookiecutter templates, which can be used to generate new projects. To configure a default cookiecutter template, add a `default_cookiecutters` dictionary to your global palm config file (`~/.pam/config.yaml`). The key is a shorthand name for the template, and the value is the cookiecutter template url. For example:

```
default_cookiecutters:
  dbt: 'https://github.com/datacoves/cookiecutter-dbt'
```

You can then use `palm new -p dbt` to generate a new dbt project using the cookiecutter template.

Recommendation: Sharing a global set of default cookiecutter templates is a great way to standardize your organization's approach to new projects. We recommend you share a set of default cookiecutter templates across your organization, and document them in your organization's documentation system.

1.1.5 Plugins

Palm plugins extend the functionality of the CLI. They usually add new commands that are specific to a particular platform or framework. A plugin could also share organization-specific functionality across multiple projects, or provide a common base for a set of commands.

Core plugins

To simplify the CLI implementation, all Palm commands come from Plugins. Within the Palm CLI repository, you will find a directory called "plugins" which contains a set of plugins. The "core" and "repo" plugins are automatically loaded when the CLI runs, these plugins provide the following functionality:

- core: Provides the core commands of Palm CLI.
- repo: Loads your custom commands from the `.palm` directory if available.

Note that the repo plugin does not provide any commands of its own, instead it provides the mechanism to load your own project-specific commands.

Installing Plugins

Palm plugins are installed as pypi packages. To install a plugin, install the pypi package for the plugin in the same python path as Palm CLI.

e.g. `pip install palm-dbt`

Configuring Plugins

Once installed, you can configure your project to use a plugin by adding the name of the plugin to the project's `.palm/config.yaml` file. See the plugin's documentation for more information on how to configure each specific plugin.

Example plugin configuration

```
plugins:
  - dbt
```

Configuring Global Plugins

You can also configure plugins that are not specific to a project. This is done by adding the name of the plugin to the `~/.palm/config.yaml` file in the user's home directory. This global configuration file will be created the first time palm is run.

Example global plugin configuration

```
plugins:
  - workflow
  - git
```

Using plugin commands

Once you have installed and configured a plugin, it's commands should be available for use within the project. To confirm this and to explore the available commands run `palm --help` to list the available commands in your project.

Keeping up to date

As with all software, plugins are likely to change over time. To keep up to date, palm provides some utility methods to check which version you are running and update to the latest version if available.

- `palm plugin versions` to list all versions of configured plugins.
- `palm plugin update --name <plugin_name>` to update a specific plugin.
- `palm plugin --help` to see all plugin utility commands.

Writing your own plugins

So, you have a set of commands that you want to re-use across projects? Or, maybe you have a set of commands you want to share with other people? Writing a plugin is a great way to contribute to the Palm CLI ecosystem.

Check out the :doc:'write-a-plugin' section to learn how to write your own plugin.

Plugin config

Some plugins require configuration to work. This configuration is stored in the `.palm/config.yaml` file in the project directory. The configuration for each plugin is stored under the `plugin_config` key at the root of the config file.

Example plugin config

```
plugin_config:
  dbt:
    prod_artifacts: path/to/artifacts/
    target: /path/to/target/
```

Plugin config is loaded into the plugin's `config` attribute. Commands can access this config by calling `environment.plugin_config(plugin_name)`.

1.1.6 Containerization

Beta Feature Note that this feature is intended to provide a basic baseline for containerization. For more complicated projects, we recommend spending some time to implement your own containerization solution. The code generated by Palm may serve as a good starting point.

Containerization (with Docker) is a pre-requisite for using Palm with your project. We understand that you may not be a Docker pro, but we can help you get started with dockerizing your project so that you can use Palm!

How to containerize your project

Palm ships with basic containerization for Python projects.

Before you start

- Make sure your project is initialized with `palm init`, the image_name configuration will be used by the containerization tool.
- You should ensure you have a .env file in your project root directory. This file should include the environment variables you need to run your project. It should also be added to your .gitignore file.
- You need to be using requirements.txt or poetry.lock to manage your dependencies. If you are using a different package management system, please open an issue on Github and we will consider adding support for it.

Use `palm containerize` to set up your project with Docker.

See `palm containerize --help` for more information.

How containerization works

Palm containerization generates the following files in your project root directory:

- Dockerfile: The Dockerfile used to build your project
- docker-compose.yml: Docker compose is used to load the .env file and volume mount your project code so that you can make changes without having to rebuild or restart the container.
- scripts/entrypoint.sh: This script is used to run your project. It is executed by the Dockerfile as it's entrypoint. This is where your dependencies are installed. Note: we do not recommend executing your project code directly in the entrypoint as this limits the flexibility of your container, instead use palm commands to run your project, this allows you to determine what each instance of the container is doing (run, test, etc.).

Implement your own containerization solution

Palm containerization is intended to provide a basic baseline for containerization for Python projects. For projects using a specific framework, or a different language, we recommend that you implement your own containerization solution. To do this, you will need to:

- Create a new containerizer command in your project (`palm scaffold command --name containerize`).
- Duplicate the templates from `palm/plugins/core/templates/containerize` and add them to your project. Make the necessary changes to support your containerization needs.
- Subclass the `Containerizer` class and override the `run()` and `package_manager()` methods.
- Open up cmd_containerize.py and implement any logic + Call YourSubclass.run() in your command.

To see an example of custom containerization, see `cmd_containerize.py` in the `palm-dbt` plugin!

Once you have implemented your containerization solution, consider releasing it to the community as a palm plugin!

1.1.7 Branding

Brand Assets

Logos

Usage of palm logos is subject to our *branding guidelines*.

Standard Color Scheme

png: 934W x 319H || png: 294W x 100H || svg

380W x 467H || 100W x 138H || svg

934W x 319H || 294W x 100H || svg

PALM

PALM

CLI framework

Dark Color Scheme (transparent)

`png: 934W x 319H || png: 294W x 100H || svg`

`380W x 467H || 100W x 138H || svg`

`934W x 319H || 294W x 100H || svg`

White Color Scheme (transparent)

`934W x 319H || png: 294W x 100H || svg`

`380W x 467H || 100W x 138H || svg`

`934W x 319H || 294W x 100H || svg`

Palm is empowering and inclusive by design, enabling Developers and Engineers With diverse skill sets to create and evolve software without homogeneous programming know-how. With palm, contributors can focus on being experts in a way that generates the most value, and skip "sweating the small stuff" of rote interface memorization. We are proud of what palm is and what it represents. These branding guidelines and associated assets are part of that representation and are intended to enhance and promote the public integrity of the project.

In plain words, we are proud of palm, and we want these branding guidelines and assets to be used freely in an open and positive way.

Basic Use Guidelines

1.1 Capitalization:

Palm is an improper (or common) noun, and should follow appropriate grammar rules. Most notably, palm is spelled with a capital "P" (Palm) as the first word in a sentence, and with a lowercase "p" (palm) otherwise.


PALM


PALM
CLI framework


PALM


PALM

Logo Usage

Palm *Logos* can be used anywhere palm software is incorporated or used as part of a project or workflow, referenced, compared or combined. Basically, if you are using or discussing palm software, we encourage you to use palm logos as you see fit. Palm logos cannot be used for projects or as part of media unrelated to palm - for example, please do not open a nightclub named "Palm" and use our logo. If you are unsure if your use violates these guidelines, drop us a line at data-analytics-team@palmetto.com and we will be happy to advise.

Swag

We are excited that you are excited about palm! so feel free to create t-shirts, hats, parade floats etc with the palm logo for personal or team use. If you intend to sell palm-branded merchandise for profit, please drop us a line at data-analytics-team@palmetto.com so we can discuss.

1.1.8 CONTRIBUTING

Contributions are welcome! Please see the **`CONTRIBUTING.md`_** file for details.

Development environment

To set up a development environment, you can use the provided **`Dockerfile`_** to build a Docker image. This image contains all the dependencies needed to run the tests and build the documentation.

Pre-commit

This project uses *pre-commit* to run a set of checks before each commit. To install the pre-commit hooks, run:

```
pre-commit install
```

1.2 Usage

Check out the *Usage* section for further information, including how to installation the project.

Note: This project is under active development.