pet Documentation

Release 0.0.1

LimeBrains

Jul 26, 2019

Contents

CHAPTER 1

Quickstart

1.1 Installation

To install **Pet**, open an interactive shell and run:

```
bash -c "$(curl -fsSL https://raw.githubusercontent.com/limebrains/pet/master/install.
↪bash)"
```

Or to specify installation directory and type of shell:

```
bash -c "shell='bash_or_zsh';install_dir='absolute_path';$(curl -fsSL https://raw.
↪githubusercontent.com/limebrains/pet/master/install.bash)"
```

1.2 Using Pet

To start using **Pet**, you need to first create a project:

```
$ pet init
```

Note: If name for new project is not passed **Pet** uses current directory name. To use custom name invoke it with `pet init -n chosen_name`

It will edit two files *start.sh* and *stop.sh* which commands are going to be executed accordingly during start of project and after closing

If you want to start project:

```
pet project_name
```

Now you are in subshell created using your standard files (like *.bashrc* or *.profile*) and *start.sh*

CHAPTER 2

User Guide

Here you can read more about how **Pet** works.

This part provides examples of use and explores all **Pet** commands and its possibilities.

2.1 Using pet

Pet can be used to:

- separate shell environments for each project you are working on
- cleaning after you work
- prepare exact shell that you need for undisturbed work
- share same shell scripts and aliases with your team
- use popular templates for working on certain tasks

2.1.1 Basic usage

This part covers basic usage of a **Pet**

Creation of project

Init

You can create project by using `pet init` command.

This leads to creating *start.sh* and *stop.sh* files in which you can add all the commands that should be run every time you start or stop project.

Pet uses current directory name as projects name if one is not provided. You can provide one with use of `-n` flag:

```
pet init -n my_awesome_project
```

Create with templates

During creation you can specify which templates to use with `-t` flag. Templates contain files that are going to be integrated to your project.

Basically it accumulates *start.sh* and *stop.sh* from every template to your new project.

To do that execute:

```
$ pet init -t first_template -t secound_template...
# or
$ pet init -t=first_template,secound_template...
```

After accumulating commands from given templates it passes editing to you with notes from which template which part of a code comes from.

Create in place

You can create project and store it's files in .pet folder in project directory by using -i flag. This can be very useful if you want to share pet project with others by adding .pet folder to repository

```
$ pet init -i
```

Editing

To edit project, task or locales you can use *edit* command.

Using edit

edit command helps you edit project or task if there is active project or edit project if non is active

```
# edits task
[project] $ pet edit task_name
# edits current project
[project] $ pet edit
# edits project
$ pet edit project_name
```

It opens accordingly task file or *start.sh* and *stop.sh* in $EDITOR

To edit locals you can use:

```
[project] $ pet edit task_name -l
# or
$ pet edit project_name -l
```

Listing

Pet provides listing of:

- available projects `pet list`
- available tasks `pet list -t` (available only when some project is active in current shell)
- archived projects `pet list -o`
- all above `pet list --tree`

Creating folders required by pet

Pet requires specific folders in *PET_FOLDER* directory, to make them you can use *recreate*

Using recreate

Recreate have to be turned without sudo. It creates required directories in *PET_FOLDER*

```
$ pet recreate
```

Stopping a project

To stop a project you can use *pet stop* or ^D, both work same way

Using stop

stop sends SIGKILL to current shell.

Commands in *stop.sh* are executed before exiting a project by using `trap 'source /path/.../stop.sh' EXIT`

```
[project] $ pet stop
```

2.1.2 Advanced usage

This part covers advanced usage of a **Pet**

Archiving a project

If you don't want to use certain project anymore but want to keep project files, you can archive it.

Project is going to be moved to *PET_FOLDER/archive* and is not going to be shown in autocomplete

Using archive

If project is not active it can be moved to archive to keep files safe.

To restore project from archive use *restore* command

```
$ pet archive project_name
$ pet restore project_name
```

To check projects in archive folder use *pet list -o*

Configuring pet

You can configure pet by using *config* command.

You can set: - editor that is going to be used to edit files used by **Pet**

- directory in which pet files are going to be stored
- files used to initialize shell (such as .bashrc or .profile)

Using config

```
$ pet config editor
```

Helps you set EDITOR variable in config file, this editor is going to be used to edit files used by **Pet**

```
$ pet config projects_folder
```

Informs you that pet stores files in directory that either is equal to *PET_FOLDER* variable that can be exported in shell profile file, if is unset uses ~/.pet folder

```
$ pet config shell
```

Helps you edit file that is going to be used to initialize new shells

Deploying auto-completion

Pet provides auto-completion for bash and zsh, to use it you have to run pet deploy (sometimes requires sudo)

Using deploy

deploy searches for possible paths to deploy completion to for shell you are using (you can choose shell type manually by using -s option)

```
$ pet deploy
```

Using local files

If you want to use relative paths or have some additional settings but eg. you share pet project in repository you can achieve that by using *local* files.

Add *.*local*.* to .gitignore to not share local files

Locals in projects

Before executing *start.sh* and *stop.sh* pet looks for local files for each of these files. If they exist they are used before and after executing *start.sh* and *stop.sh*.

Workflow: *start.local.entry.sh* -> *start.sh* -> *start.local.exit.sh*

work

stop.local.entry.sh -> *stop.sh* -> *stop.local.exit.sh*

To edit this files you can use:

```
$ pet edit project_name -l
```

Locals in tasks

Before executing task file pet looks for *task_name.local.entry.sh* and *task_name.local.exit.sh*. If they exits they are used before and after executing task.

Workflow: *start.local.entry.sh* -> *task* -> *start.local.exit.sh*

To edit this files you can use:

```
[project] $ pet edit task_name -l
```

Register

Registers current directory as folder with project configuration files for pet if found all required files:

- *start.sh*
- *stop.sh*
- tasks (directory)

```
$ pet register
```

You can add it under specific name different than directory name by passing parameter with -n flag

```
$ pet register -n project_name
```

This might be useful if you want to share same environment in shell for each member of a project - just add it to git repository with your project.

Whenever someone makes changes you are going to be using same environment.

This is accomplished by adding symbolic link to folder with projects

Deletion

To delete project, task or locks you can use *remove* command.

Using remove

remove command if given name removes task if there is active project or removes project if non is active, also can be used to delete all lock files that prevent from initializing project too many times.

```
[project] $ pet remove task_name
# or
$ pet remove project_name
# delete locks
$ pet remove -l
```

Renaming

To rename project or task you can use *rename* command.

Using rename

rename command renames task if there is active project or renames project if non is active

```
[project] $ pet rename old_task_name new_task_name
# or
$ pet rename old_project_name new_project_name
```

Restoring project from archive

If you want to restore project that you previously put in archive you can use *restore* command

Using restore

To restore project from archive use *restore* command

```
$ pet restore project_name
```

To check projects in archive folder use *pet list -o*

Running a task outside of project

To run task outside of a project you can use *run* command

Using run

Run command runs task from a project in projects environment.

It is run in subshell.

To stay in shell after task is completed you can use *-i* flag which stands for interactive mod

```
$ pet run project_name task_name
# or
$ pet run project_name task_name -i
```

Creating task

To create task you need to activate project first, than use *task* command.

Using task

Running *task* command will make a script that will be available during use of a project.

You can specify what type of a file it's going to be, but name of a task is understood as name of a file without extension.

If extension is not provided it will create *.sh* file.

You have to choose is this task going to be normal or local by using either -s/ –save or -l/ –local flag.

Local tasks are stored with additional '.local' in their names

You might want to use local tasks if you are sharing pet project in repository.

```
[project] $ pet task task_name -s
# or
[project] $ pet task task_name.extension -s
# or
[project] $ pet task task_name -l
# or
[project] $ pet task task_name.extension -l
```

This opens task file in $EDITOR to let you edit it.

You can change file extension freely

Task without alias

If you don't want to create alias to task eg. because it have a name of shell command you can use -a flag.

```
[project] $ pet task task_name -s -a
```

Running task

To run a task you can do it from within project:

```
[project] $ pet task_name
# or by using alias (if -a flag was not used) during every next invocation of a␣
↪project
[project] $ task_name
```

To run it from outside of a project you have to perform:

```
$ pet run project_name task_name
```

Templates

Provide faster creation of projects

Templates are stored in *PET_FOLDER/templates*

Using templates

You can use templates during initialization of a project by adding a *-t* flag before each templates name.

```
$ pet init -t first_template -t secound_template...
# or
$ pet init -t=first_template,secound_template...
```

2.2 Modules

Look into source code of pet

2.2.1 Business logic

2.2.2 Pet exceptions

2.2.3 Pet utils

pet.utils.**makedirs**(*exists_ok=False*, **args*, ***kwargs*)

CHAPTER 3

Indices and tables

- genindex

Python Module Index

p

Index

M

P