CONTENTS

// Tutorial //

How to Use Editors, Regex, and Hooks with Z-shell

Published on May 2, 2023

Developer Education Write for DO

By Alvin Charity

ter / Developer / Musician

Introduction

`zsh`, otherwise known as the Z shell, was created by Paul Falstad in 1990 to provide an alternative to the Bourne Shell. `zsh` provides many customization options that are not available in other shells, making the Z shell a feature rich alternative to `sh` and `bash`, as well as lesser used shells like the Korn shell (`ksh`) and the C shell (`csh`). `zsh` has become popular for command line users who want to tailor their shell experience specifically to their needs.

This article will cover a sampling of the features offered by `zsh`, including using `zsh` to emulate other shells, editing on the command line using built in `zsh` features alongside the external tools `vidir` and `vipe`, using regular expressions in `zsh` commands, and setting up hooks to automate shell behavior.

Prerequisites

Before getting started with the code in this article, make sure that you have a terminal emulator installed on your system and your `zsh --version` is `5.8.1` or higher.

If you're using Linux and your `zsh` version is lower than `5.8.1` you can update by using your system package manager. For example, using `apt` on Ubuntu will install `zsh` version `5.9`.

MacOS users will need to install an updated version of `zsh` which can be done via the `homebrew` tool:

```
$ brew update && brew install zsh
```
Copy

To use `vidir` and `vipe` you must install the `moreutils` package on your machine. This package is available for Linux via joeyh.name/code/moreutils/, or installed with your package managers on Debian, Ubuntu, and Arch. `Moreutils` is also available on MacOS via `homebrew` or `macports`.

Finally, this article assumes that you have some familiarity with command-line tools, and editing files in the terminal. Familiarity with `zsh` dotfiles, including the `.zshrc` file is required for some of the examples below.

Now, let's get started!

Compatibility with Other Shells

While `zsh` is one of several popular command line shells, the basic usage of `zsh` does not differ greatly from other shells. Additionally, `zsh` includes certain compatibility options with other shells for those times you prefer specific options offered by other shells, or you're testing `zsh` out for the first time.

The tools provided by `zsh` to mimic the behavior of other shells fall into two categories:

1. Using shell options (`setopt`)
2. Using the `emulate` Builtin

Let's take a look at what you can due using these commands, starting with `setopt`

1. Using Shell Options with `setopt`

The `setopt` tool offers many options for customizing your shell, but for the purposes of this article, we will cover only options that are most useful for making your `zsh` experience a bit more comfortable when coming from another environment. The options below all focus on bringing different behavior to the `zsh` shell whenever you need it.

But first, how do you even know what options are active in your shell? Running `setopt` without an argument will display a list of options you have enabled. While this is helpful, you may have to do a little digging to find a full list of options available in the `zsh` shell. However, there is a way to see a full list of available options directly in your shell by using `ksh_option_print`.

ksh_option_print

The `ksh_option_print` option lets you change the list of options printed with `setopt` from showing only enabled options to showing each option and a corresponding `on` or `off` column for each.

```
# setopt kshoptionprint will also work
setopt ksh_option_print
```

Here's an abridged list of options from `setopt` with an on/off indicator for each option.

```
$ setopt
```

Copy

```
noaliases   off
allexport   on
noalwayslastprompt   off
alwaystoend   on
appendcreate   off
noappendhistory           off
autocd   on
autocontinue   off
noautolist   off
noautomenu   off
autonamedirs   off
noautoparamkeys   off
noautoparamslash   off
autopushd   on
noautoremoveslash   off
autoresume   off
nobgnice   off
braceccl   off
bsdecho   on
```

bsd_echo

By default `zsh` will print newline (`\n`) or tab (`\t`) characters as non-visible, but there may be times where you need to have the literal backslash escaped character present in a string. For these times, you can activate the `bsd_echo` option.

```
setopt bsd_echo
# prints tab and newline characters
echo "\t\n"
```

```
unsetopt bsd_echo

# will print the literal "\t\n"
echo "\t\n"
```

csh_junkie_loops

The `csh_junkie_loops` option offers a more terse syntax for looping over lists or arrays. Activating this option lets you use loops without requiring the `do` keyword before executing commands on each item in the loop. Additionally, the `csh_junkie_loops` option allows you to swap the `done` keyword for `end`, which saves a keystroke.

While this may not seem like a huge improvement, the shortened syntax can help you interact with your shell more quickly over time by eliminating the overhead of remembering the keywords necessary to create a syntactically correct loop in `zsh`.

The example below:

```
# the for command requires "do" before any actions
# can be performed on the list of items
# a traditional zsh loop
for f in 1 3 4 5; do print $f; done
```

With the `csh_junkie_loops` option enabled, you can use a shorter syntax to achieve the same result:

```
setopt csh_junkie_loops
for f in 1 3 4 5; echo $f; end
$ 1
3
4
5
```

csh_nullcmd

When clearing the contents of a file in your terminal, you may use a command similar to `>file.txt` which will redirect an empty string to replace the contents of your file. If you are working with sensitive data, or you want to avoid overwriting files with empty data, you can activate the `csh_nullcmd` option to prevent file redirections from occurring without a preceding command.

After activating this option, any redirection that does not follow a command will result in a error. For example

```
$ setopt csh_nullcmd
$
$ # attempting to overwrite file.txt will create an error
$ >file.txt


zsh: redirection with no command
```

Copy

You can disable this option by entering `unsetopt chs_nullcmd` in your terminal.

ksh_arrays

In `zsh`, arrays are One-indexed, meaning that you must use `$array[1]` to retrieve the first item. This behavior is counterintuitive for developers who are used to using programming languages that include zero-indexed arrays.

Arrays in `ksh` allow you to use zero-indexing which mimics the behavior of many programming languages. This option also requires that you use curly braces to access array elements, instead of using square brackets, as with the default `zsh` behavior in which the curly braces are optional.

```
declare -a kitchen_items=(plates cutlery oven sink)
print ${kitchen_items[1]}
# plates
```

With the `ksh_arrays` option set, `$kitchen_items[1]` references “cutlery” instead of “plates”

```
setopt ksh_arrays
declare -a kitchen_items=(plates cutlery oven sink)

print ${kitchen_items[1]}
# cutlery
```

sh_word_split

By de`zsh` does not split multi-word variables into individual elements like `sh` and `ksh` shells. This means that a variable containing three words separated with spaces will be

treated as a single item, rather than three individual words.

```
kitchen_items="plates cutlery oven sink"

# without sh_word_split
for word in $kitchen_items; do
    print "$word"
end
```

To force `zsh` to mimic the behavior of `sh` by splitting on spaces, use `setopt sh_word_split`

```
$ setopt sh_word_split
```

Copy

Now using let's print the contents of `$kitchen_items` using slightly different syntax:

```
kitchen_items="plates cutlery oven sink"
for word in $kitchen_items[@]; do
    print "$word"
end
# the loop above will print the following text:
# plates
# cutlery
# oven
# sink
```

The examples above can be used on-the-fly in your shell, or for more permanent usage of `setopt` you can add any of the above `setopt` commands to your `.zshrc` file to make these options available in any interactive shell session.

However, if you find yourself in need of a different way to mimic the behavior of other shells, let's take a look at the `emulate` command.

2. Using the `emulate` Command

To emulate another shell in `zsh`, you can use the emulate builtin command. For example, to emulate the Bash shell, you can use the following command:

```
emul    bash
```

This will enable `zsh` to behave like the Bash shell in most respects. You can then run Bash scripts and commands in `zsh`, and they should work as expected.

To emulate other shells, such as the Bourne or Korn shells, you can use the corresponding name in place of bash in the emulate command. For example, to emulate the Korn shell, you can use the following command:

```
emulate ksh
```

If you have a command that you would like use in a `sh` emulation mode, you can use

```
emulate sh
```

Once set, you can print the current emulation mode by typing `emulate` in your terminal.

The `emulate` command also allows you to apply “sticky” emulation to functions by using the `-c` option. This means the emulated command will be in scope for any command that executes later in this function, after the `emulate` command is executed. Whenever the function is executed this emulation will be active.

```
function hello_world_sh () {
    emulate sh -c  "print hello world";
}
```

The `emulate sh -c` command can be useful for creating functions that require POSIX compatibility while still taking advantage of the `zsh` environment.

Editing on the Fly

During your daily work you may need to edit many kinds of files - configuration for the app you’re building or if you’re only checking an item off a to-do list. Being able to edit these files quickly is a benefit to working in the terminal in general, and is especially useful when using a shell as customizable as `zsh`.

There are several tools available for `zsh` that will allow you to edit various aspects of your environment within `zsh` seamlessly. Starting with the core of the interactive terminal, the zsh Line Editor.

The zsh Line Editor, or ZLE is the editor that you use with `zsh` to enter and modify commands on the command line in your terminal. All shells have a similar line editor, and in most cases you won't need to modify its behavior.

ZLE activates automatically when `zsh` is loaded, but it also exists as a `zsh` module called `zsh/ZLE`.

In `zsh`, the ZLE (zsh Line Editor) is the built-in line editing interface that allows you to edit the command line and navigate the command history. ZLE provides a number of keybindings and functions that you can use to interact with the command line, such as moving the cursor, deleting text, and accessing the command history.

If you are using `zsh` in an interactive shell, ZLE is enabled by default since it is the core of the interactive functionality of `zsh`. You can use the following command to verify the `zle` option is set:

```
# This command will print "ZLE: on" if the zsh Line Editor is enabled.
[[ -o zle ]] && echo "zle: on"
```

The main use for directly interacting with ZLE is through setting your default keybindings for your interactive shell environment. These options can allow you to use keybindings specific to your preferred editor, either `emacs` or `vi`. The `zsh` users guide provides a detailed explanation of the differences between these two editing modes.

If you prefer the emacs keybindings, enter the following command in your terminal.

```
bindkey -e
```

To enable vi keybindings, use the following command:

```
bindkey -v
```

Now that you learned a little bit about the zsh Line Editor, let's see how you can use it with the `zed` editor.

Zed

`Zed` uses ZLE to let you edit scripts and functions on the command line without having to fire up an editor. This is great for quick edits as you navigate your file system, but can also be used as a basic editor for any general editing command. `Zed` exists as a short

script so you won't find the same bells and whistles as a full-featured editor, however simplicity is what makes `zed` and effective tool.

To get started, type `zed` in your terminal to check it is installed on your system.

```
zed
```

If the `zed` command is not available, you can grab the code from https://github.com/zsh-users/zsh/blob/master/Functions/Misc/zed and save it as an executable file, or wrap it in a function to be sourced.

Once `zed` is set up in your environment, typing `zed` in your terminal will create a new , where you can edit the command line as needed. When you save and exit the editor, the edited command line will be executed by `zsh`.

To edit a specific function in your `zsh` environment, you can use `zed -f` like so

```
zed -f precmd
```

Note that the changes made to your function with `zed` will only exist until you reload your configuration files via `source .zshrc` or similar command.

Vared

`vared` allows you to modify the value of a variable using your preferred text editor, rather than using the set or export commands to set the value manually on the command line. To use the `vared` command, you first need to specify the name of the variable that you want to edit. For example, to edit the value of your `EDITOR` environment variable, you can use the following command:

```
vared EDITOR
```

This will open the value of the `EDITOR` environment variable using `ZLE`, where you can edit the value as needed. When you are finished editing your variable you can use `ctrl-c` to save the changes and return to the command line.

Editing Directory Names with `vidir`

`Vidir` vs you to edit the names of directories (or files) in your default editor. This is useful for quickly modifying directory names visually, rather than relying on pattern

matching or loops. For example, to edit the names of all items in your `$HOME` directory, you can use the following command

```
vidir "$HOME"
```

`Vidir` will create a temporary file containing the names of each item in your `$HOME` directory. After you make your changes and save the file, `vidir` will update the name of any file or folder you modified.

You can also use `Vidir` to change the name of a single file name by passing your chosen filename to `vidir` as an argument instead of a directory. For example

```
vidir /path/to/file.txt
```

Will open `file.txt` in your default editor, where you can modify the filename as you please.

Using Editors in Pipelines with Vipe

It is sometimes useful to modify the contents of a pipe in your default editor, rather than rely on tools like `awk` or `sed` to perform actions on text within a pipeline. The `vipe` command can be inserted into any pipe and will allow you to edit the result of the previous command visually. This is especially useful for times when you aren't sure what the result of the command will look like, and therefore you are unable to apply a specific command.

The following example uses `vipe` to modify the contents of the preceding `curl` command, and then saves the result to a file called `example_modified.txt`

```
curl "https://example.com" | vipe > example_modified.txt
```

Regular Expressions

`zsh` offers an extensive set of glob operators, a form of expansion, that can be overwhelming if you are coming from a more basic shell. Luckily `zsh` offers the ability to use regular expressions for pattern matching throughout your shell, which may be a better option if you find globbing to be confusing or unintuitive.

I tend to turn globbing off entirely in my environment by using the `noglob` shell option, but this is not required:

```
setopt noglob
```

Regex in Test Blocks

`zsh` provides basic Regular expression matching via the `=~` comparison operator. This allows you to use regular expressions to match text when executing `if` blocks. For example, let's check if `$variable` contains the text "txt" or "zsh":

```
variable="file.txt"

if [[ "$variable" =~ (txt|zsh) ]];
    then
        print "variable contains txt or zsh";
    else
        print "variable does not contain txt or zsh"
fi
```

Using zsh/Regex

The `zsh/regex module` allows you to use a flag to enable regular expressions in test conditions. This flag uses POSIX extended regular expressions to match portions of text.

To load `zsh` modules into your environment, use the `zmodload` command:

```
zmodload zsh/regex
```

Once loaded, you can use the `-regex-match` flag in test blocks to apply regular expressions. For example, you can shorten the `if` block from the example above to a `test` one line command. When a match is found, the matching portion of the string is added to the environment variable `$MATCH`.

```
zmodload zsh/regex

test "txt" -regex-match (txt|zsh) && echo "contains txt or zsh" || print "do
```

And you can check for the specific match by printing the `MATCH` environment variable

```
$ p    $MATCH
$
$ txt
```

Copy

Using `zsh/PCRE`

As with `zsh/regex` you can load the `zsh/prce` module via the `zmodload` command:

```
# note: pcre must be lowercase for the module to load
zmodload zsh/pcre
```

The `zsh/pcre` comes with three commands: `pcre_compile`, `pcre_study`, and `pcre_match`.

- **pcre_compile** Compiles a perl-compatible regular expression.
  - The options for this command match the options for PCREs

- **pcre_match** A standalone command to match strings against PCREs. Usage

```
pcre_compile -m "bark$"
string="dog says bark"
pcre_match -b -- $string
print $MATCH
```

Additionally, zsh/pcre offers a test condition called `-pcre-match`, which matches against strings, similar to the `-regex-match` flag available in the `zsh/regex` module mentioned above. The following code uses the `-pcre-match` flag to match a string against a perl-compatible regular expression.

```
test "txt" -regex-match (txt|zsh) && print "contains txt or zsh" || print "c
```

Respond to Events with Hooks

`zsh` Hooks are special functions that are automatically executed at specific points in the shell's execution, such as when a command is entered at the prompt or when the shell starts up. Hooks allow you to perform certain actions or tasks at these points, such as modifying the command before it is executed or displaying a message to the user.

In `zsh`, the `chpwd` function is a special function that is automatically executed whenever the current working directory is changed. This allows you to perform certain actions or tasks whenever the user changes directories.

To use the `chpwd` function, you first need to define it in your `.zshrc` file. To define the `chpwd` function, add the following lines to your configuration file:

```
chpwd () {
  # Your commands and actions go here
}
```

You could use the `chpwd` function to update the directory in the command prompt, or to display a message indicating the new working directory.

Once you have defined the `chpwd` function in your configuration file, it will be automatically executed whenever the current working directory is changed. You can test it by changing directories in your `zsh` session and observing the effects of the `chpwd` function.

Periodic

Writing a single function called periodic will run the containing commands every n seconds, where the seconds are stored in the environment variable `$PERIOD`. For example, say you want to run a function that downloads my bookmarks every six hours. To do this, you can use the following code.

The time is set using the environment variable `$PERIOD` to set the frequency in which the function runs.

```
periodic() {

}
```

precmd

This does not run when the terminal is redrawn. The redrawing happens when a background process sends a notification to your current prompt.

For a more concrete example, my `precmd` function contains a command that stores the current directory in a file. This allows me to open a new terminal in the most recently used direct[illegible]his is especially useful if you are testing your environment and have to restart your terminal to see the results, or if your terminal accidentally crashes while working on an important project.

```
precmd () {
  pwd >"$HOME/.zsh_reload_directory.txt" &
}
```

This appends the current directory to a file called `.zsh_reload.txt` in my `$HOME` directory. To automatically navigate to this directory when your terminal loads, add the following command elsewhere in your `.zshrc` file

```
# create the precmd function
precmd () {
  pwd >|"$HOME/.zsh_reload_directory.txt" &
}

# add the navigation command outside of the precmd function
cd "$HOME/.zsh_reload_directory.txt"
```

preexec

Executed after the command is read, but before the command prompt is displayed.

Typical use for the `precmd` function is to set the `$PS1` command prompt variable with useful information, such as the time of your last executed command, however you are not limited to updating your prompt.

zshaddhistory

`zshaddhistory` is a special function that is automatically executed whenever a command is entered at the prompt. This allows you to perform certain actions or tasks whenever a command is entered, such as logging the command or modifying the command before it is executed.

To use `zshaddhistory` you can define it as a function in your `.zshrc` file:

```
zshaddhistory () {
  # Your commands and actions go here
}
```

You could use the `zshaddhistory` function to log the command to a file, or to modify the command before it is executed.

Once you have defined the `zshaddhistory` function in your configuration file, it will be automatically executed whenever a command is entered at the prompt. You can test it by

entering a command at the prompt and observing the effects of the `zshaddhistory` function.

Adding New Hooks

To use the `add-zsh-hook` function, you first need to define a custom hook function that contains the commands and actions that you want to execute at the specified point. For example, you could define a hook function that logs the commands entered at the prompt, like this:

```
log_commands () {
  print "$(date): $1" >> ~/.zsh_command_log
}
```

Once you have defined your custom hook function, you can use the `add-zsh-hook` function to attach it to an existing `zsh` hook. For example, to attach the `log_commands` function to the `zshaddhistory` hook, which is executed whenever a command is entered at the prompt, you can use the following command:

```
add-zsh-hook zshaddhistory log_commands
```

This will cause the `log_commands` function to be executed whenever a command is entered at the prompt, and the command will be logged to the `~/.zsh_command_log` file.

```
hooks-define-hook
```

In `zsh`, the hooks-define-hook function is used to define a custom hook. `zsh` hooks are special functions that are automatically executed at specific points in the shell's execution, such as when a command is entered at the prompt or when the shell starts up. Defining a custom hook allows you to create your own points in the shell's execution at which to execute your own commands or actions.

To use the `hooks-define-hook` function, you first need to decide when and where you want your custom hook to be executed. This will determine the name and arguments of the hook. For example, if you want your hook to be executed whenever a command is entered at the prompt, you could use the `zshaddhistory` hook, which is executed with the command as its only argument.

Once you have decided on the name and arguments of your hook, you can define the hook using the `hooks-define-hook` function. For example, to define a hook that is executed

whenever a command is entered at the prompt and logs the command to a file, you could use the following command:

```
hooks-define-hook zshaddhistory log_command
```

This will define a hook named `zshaddhistory` that takes a single argument (the command entered at the prompt).

Once you have defined the hook, you can use the `add-zsh-hook` function to attach a function to the hook. This function will be executed whenever the hook is triggered.

Conclusion

In this article you learned about many ways you can customize your workflow with the Z shell by using editors on the fly, applying regex to match variables and other text, and setting up hooks in your environment to automatically respond to events in your shell. This article only touches on a portion of the customization options available through `zsh` - for more information about the shell visit the official documentation at https://zsh.sourceforge.io/.

About the authors

Alvin Charity Author

Writer / Developer / Musician

Still looking for an answer? Ask a question

Search for more help

Was this helpful? Yes No

Comments

Leave a comment

Leave a comment...

This textbox defaults to using **Markdown** to format your answer.

You can type **!ref** in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment

Popular Topics

Ubuntu

Linux Basics

JavaScript

Python

MySQL

Docker

Kubernetes

All tutorials →

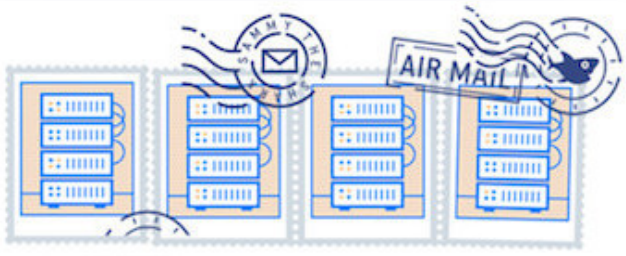
Free Managed Hosting →

Congratulations on unlocking the whale ambience easter egg! Click the whale button in the bottom left of your screen to toggle some ambient whale noises while you read.

Thank you to the Glacier Bay National Park & Preserve and Merrick079 for the sounds behind this easter egg.

Interested in whales, protecting them, and their connection to helping prevent climate change? We recommend checking out the Whale and Dolphin Conservation.

Reset easter egg to be discovered again / Permanently dismiss and hide easter egg



Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

Sign up →

Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

Learn more →

Become a contributor

You get paid; we donate to tech nonprofits.

Learn more →

Featured on Community

Kubernetes Course · Learn Python 3 · Machine Learning in Python

Getting started with Go · Intro to Kubernetes

DigitalOcean Products

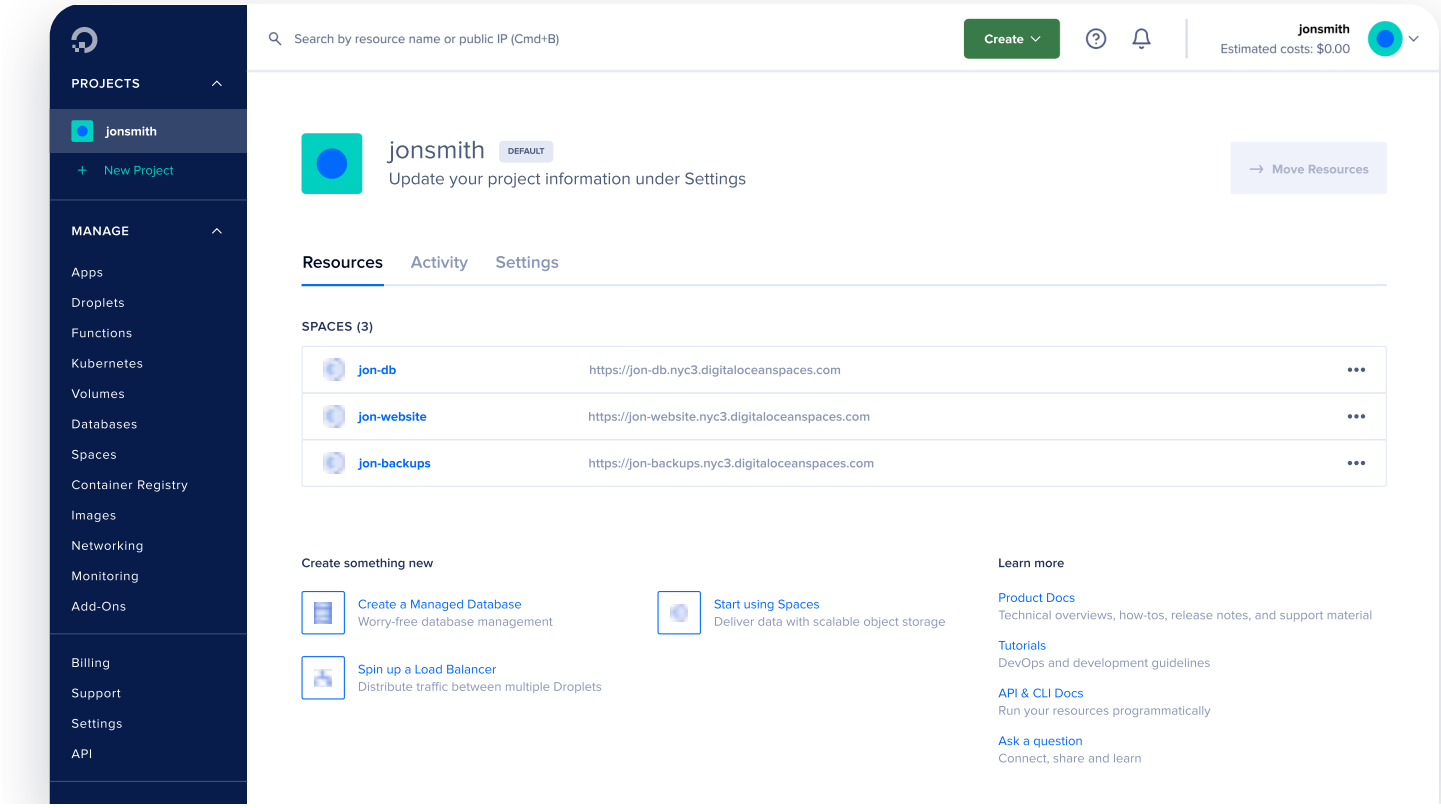
Cloudways · Virtual Machines · Managed Databases · Managed Kubernetes

Block Storage · Object Storage · Marketplace · VPC · Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

Learn more →



Get started for free

Enter your email to get $200 in credit for your first 60 days with DigitalOcean.

Email address

Send My Promo

New accounts only. By submitting your email you agree to our Privacy Policy.

Company

Products

Community

Solutions

Contact