##### **Pour plus d’infos :**

Creating a Command

Commands are defined in classes extending Command. For example, you may want a command to create a user:

Copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

// src/Command/CreateUserCommand.php

namespace App\Command;

use Symfony\Component\Console\Attribute\AsCommand;

use Symfony\Component\Console\Command\Command;

use Symfony\Component\Console\Input\InputInterface;

use Symfony\Component\Console\Output\OutputInterface;

// the name of the command is what users type after "php bin/console"

#[AsCommand(name: 'app:create-user')]

class CreateUserCommand extends Command

{

protected static $defaultName = 'app:create-user';

protected function execute(InputInterface $input, OutputInterface $output): int

{

// ... put here the code to create the user

// this method must return an integer number with the "exit status code"

// of the command. You can also use these constants to make code more readable

// return this if there was no problem running the command

// (it's equivalent to returning int(0))

return Command::SUCCESS;

// or return this if some error happened during the execution

// (it's equivalent to returning int(1))

// return Command::FAILURE;

// or return this to indicate incorrect command usage; e.g. invalid options

// or missing arguments (it's equivalent to returning int(2))

// return Command::INVALID

}

}

#

Configuring the Command

You can optionally define a description, help message and the input options and arguments by overriding the configure() method:

Copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

// src/Command/CreateUserCommand.php

// ...

class CreateUserCommand extends Command

{

// the command description shown when running "php bin/console list"

protected static $defaultDescription = 'Creates a new user.';

// ...

protected function configure(): void

{

$this

// the command help shown when running the command with the "--help" option

->setHelp('This command allows you to create a user...')

;

}

}

Tip

Defining the $defaultDescription static property instead of using the setDescription() method allows to get the command description without instantiating its class. This makes the php bin/console list command run much faster.

If you want to always run the list command fast, add the --short option to it (php bin/console list --short). This will avoid instantiating command classes, but it won't show any description for commands that use the setDescription() method instead of the static property.

The configure() method is called automatically at the end of the command constructor. If your command defines its own constructor, set the properties first and then call to the parent constructor, to make those properties available in the configure() method:

Copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

// ...

use Symfony\Component\Console\Command\Command;

use Symfony\Component\Console\Input\InputArgument;

class CreateUserCommand extends Command

{

// ...

public function \_\_construct(bool $requirePassword = false)

{

// best practices recommend to call the parent constructor first and

// then set your own properties. That wouldn't work in this case

// because configure() needs the properties set in this constructor

$this->requirePassword = $requirePassword;

parent::\_\_construct();

}

protected function configure(): void

{

$this

// ...

->addArgument('password', $this->requirePassword ? InputArgument::REQUIRED : InputArgument::OPTIONAL, 'User password')

;

}

}

#

Registering the Command

In PHP 8 and newer versions, you can register the command by adding the AsCommand attribute to it:

Copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

// src/Command/CreateUserCommand.php

namespace App\Command;

use Symfony\Component\Console\Attribute\AsCommand;

use Symfony\Component\Console\Command\Command;

// the "name" and "description" arguments of AsCommand replace the

// static $defaultName and $defaultDescription properties

#[AsCommand(

name: 'app:create-user',

description: 'Creates a new user.',

hidden: false,

aliases: ['app:add-user']

)]

class CreateUserCommand extends Command

{

// ...

}

If you can't use PHP attributes, register the command as a service and tag it with the console.command tag. If you're using the default services.yaml configuration, this is already done for you, thanks to autoconfiguration.

#

Running the Command

After configuring and registering the command, you can run it in the terminal:

$ php bin/console app:create-user

As you might expect, this command will do nothing as you didn't write any logic yet. Add your own logic inside the execute() method.

#

Console Output

The execute() method has access to the output stream to write messages to the console:

Copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

// ...

protected function execute(InputInterface $input, OutputInterface $output): int

{

// outputs multiple lines to the console (adding "\n" at the end of each line)

$output->writeln([

'User Creator',

'============',

'',

]);

// the value returned by someMethod() can be an iterator (https://secure.php.net/iterator)

// that generates and returns the messages with the 'yield' PHP keyword

$output->writeln($this->someMethod());

// outputs a message followed by a "\n"

$output->writeln('Whoa!');

// outputs a message without adding a "\n" at the end of the line

$output->write('You are about to ');

$output->write('create a user.');

return Command::SUCCESS;

}

Now, try executing the command:

$ php bin/console app:create-user

User Creator

============

Whoa!

You are about to create a user.

#

Output Sections

The regular console output can be divided into multiple independent regions called "output sections". Create one or more of these sections when you need to clear and overwrite the output information.

Sections are created with the ConsoleOutput::section() method, which returns an instance of ConsoleSectionOutput:

Copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

// ...

use Symfony\Component\Console\Output\ConsoleOutputInterface;

class MyCommand extends Command

{

protected function execute(InputInterface $input, OutputInterface $output): int

{

if (!$output instanceof ConsoleOutputInterface) {

throw new \LogicException('This command accepts only an instance of "ConsoleOutputInterface".');

}

$section1 = $output->section();

$section2 = $output->section();

$section1->writeln('Hello');

$section2->writeln('World!');

// Output displays "Hello\nWorld!\n"

// overwrite() replaces all the existing section contents with the given content

$section1->overwrite('Goodbye');

// Output now displays "Goodbye\nWorld!\n"

// clear() deletes all the section contents...

$section2->clear();

// Output now displays "Goodbye\n"

// ...but you can also delete a given number of lines

// (this example deletes the last two lines of the section)

$section1->clear(2);

// Output is now completely empty!

return Command::SUCCESS;

}

}

Note

A new line is appended automatically when displaying information in a section.

Output sections let you manipulate the Console output in advanced ways, such as displaying multiple progress bars which are updated independently and appending rows to tables that have already been rendered.

#

Console Input

Use input options or arguments to pass information to the command:

Copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

use Symfony\Component\Console\Input\InputArgument;

// ...

protected function configure(): void

{

$this

// configure an argument

->addArgument('username', InputArgument::REQUIRED, 'The username of the user.')

// ...

;

}

// ...

public function execute(InputInterface $input, OutputInterface $output): int

{

$output->writeln([

'User Creator',

'============',

'',

]);

// retrieve the argument value using getArgument()

$output->writeln('Username: '.$input->getArgument('username'));

return Command::SUCCESS;

}

Now, you can pass the username to the command:

$ php bin/console app:create-user Wouter

User Creator

============

Username: Wouter

See also

Read Console Input (Arguments & Options) for more information about console options and arguments.

#

Getting Services from the Service Container

To actually create a new user, the command has to access some services. Since your command is already registered as a service, you can use normal dependency injection. Imagine you have a App\Service\UserManager service that you want to access:

Copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

// ...

use App\Service\UserManager;

use Symfony\Component\Console\Command\Command;

class CreateUserCommand extends Command

{

private $userManager;

public function \_\_construct(UserManager $userManager)

{

$this->userManager = $userManager;

parent::\_\_construct();

}

// ...

protected function execute(InputInterface $input, OutputInterface $output): int

{

// ...

$this->userManager->create($input->getArgument('username'));

$output->writeln('User successfully generated!');

return Command::SUCCESS;

}

}

#

Command Lifecycle

Commands have three lifecycle methods that are invoked when running the command:

initialize() (optional)

This method is executed before the interact() and the execute() methods. Its main purpose is to initialize variables used in the rest of the command methods.

interact() (optional)

This method is executed after initialize() and before execute(). Its purpose is to check if some of the options/arguments are missing and interactively ask the user for those values. This is the last place where you can ask for missing options/arguments. After this command, missing options/arguments will result in an error.

execute() (required)

This method is executed after interact() and initialize(). It contains the logic you want the command to execute and it must return an integer which will be used as the command exit status.

#

Testing Commands

Symfony provides several tools to help you test your commands. The most useful one is the CommandTester class. It uses special input and output classes to ease testing without a real console:

Copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

// tests/Command/CreateUserCommandTest.php

namespace App\Tests\Command;

use Symfony\Bundle\FrameworkBundle\Console\Application;

use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;

use Symfony\Component\Console\Tester\CommandTester;

class CreateUserCommandTest extends KernelTestCase

{

public function testExecute()

{

$kernel = self::bootKernel();

$application = new Application($kernel);

$command = $application->find('app:create-user');

$commandTester = new CommandTester($command);

$commandTester->execute([

// pass arguments to the helper

'username' => 'Wouter',

// prefix the key with two dashes when passing options,

// e.g: '--some-option' => 'option\_value',

]);

$commandTester->assertCommandIsSuccessful();

// the output of the command in the console

$output = $commandTester->getDisplay();

$this->assertStringContainsString('Username: Wouter', $output);

// ...

}

}

If you are using a single-command application, call setAutoExit(false) on it to get the command result in CommandTester.