Monofony

Oct 22, 2020

Contents

Monofony is a project based on Symfony Framework and Sylius.

Note: This documentation assumes you have a working knowledge of the Symfony Framework. If you're not familiar with Symfony, please start with reading the Quick Tour from the Symfony documentation.

CHAPTER 1

The Book

Here you will find all the concepts used in the Monofony platform. *The Book* helps to understand how Monofony works.

1.1 The Book

Here you will find all the concepts used in Monofony. The Books helps to understand how Monofony works.

1.1.1 Architecture

The key to understanding principles of Sylius internal organization. Here you will learn about the Resource layer, state machines, events and general non e-commerce concepts adopted in the platform, like E-mails or Fixtures.

Architecture

Note: This section is based on the great Sylius documentation.

Before we dive separately into every Monofony concept, you need to have an overview of how our main application is structured. In this chapter we will sketch this architecture and our basic, cornerstone concepts, but also some supportive approaches, that you need to notice.

Architecture Overview

Before we dive separately into every Monofony concept, you need to have an overview of how our main application is structured.

Fullstack Symfony

Monofony is based on Symfony, which is a leading PHP framework to create web applications. Using Symfony allows developers to work better and faster by providing them with certainty of developing an application that is fully compatible with the business rules, that is structured, maintainable and upgradable, but also it allows to save time by providing generic re-usable modules.

Learn more about Symfony.

Doctrine

Doctrine is a family of PHP libraries focused on providing data persistence layer. The most important are the object-relational mapper (ORM) and the database abstraction layer (DBAL). One of Doctrine's key features is the possibility to write database queries in Doctrine Query Language (DQL) - an object-oriented dialect of SQL.

To learn more about Doctrine - see their documentation.

Twig

Twig is a modern template engine for PHP that is really fast, secure and flexible. Twig is being used by Symfony.

To read more about Twig, go here.

Third Party Libraries

Monofony uses a lot of libraries for various tasks:

- Sylius for routing, controllers, data fixtures, grids
- KnpMenu - for backend menus
- Imagine for images processing, generating thumbnails and cropping
- Pagerfanta for pagination
- Winzou State Machine - for the state machines handling

Fixtures

Fixtures are used mainly for testing, but also for having your website in a certain state, having defined data - they ensure that there is a fixed environment in which your application is working.

Note: The way Fixtures are designed in Monofony is well described in the FixturesBundle documentation.

What are the available fixtures in Monofony?

To check what fixtures are defined in Monofony run:

```
$ php bin/console sylius:fixtures:list
```

How to load Monofony fixtures?

The recommended way to load the predefined set of Monofony fixtures is here:

```
$ php bin/console sylius:fixtures:load
```

What data is loaded by fixtures in Monofony?

All files that serve for loading fixtures of Monofony are placed in the `App/Fixture/*` directory.

And the specified data for fixtures is stored in the `config/packages/sylius_fixtures.yaml` file.

- *Architecture Overview*
- *Fixtures*
- *Architecture Overview*
- *Fixtures*

1.1.2 Users

This chapter will tell you more about the way Sylius handles users, customers and admins.

Users

Before we dive separately into every Monofony concept, you need to have an overview of how our main application is structured. In this chapter we will sketch this architecture and our basic, cornerstone concepts, but also some supportive approaches, that you need to notice.

AdminUser

The **AdminUser** entity extends the **User** entity. It is created to enable administrator accounts that have access to the administration panel.

How to create an AdminUser programmatically?

The **AdminUser** is created just like every other entity, it has its own factory. By default it will have an administration **role** assigned.

```
/** @var AdminUserInterface $admin */
$admin = $this->container->get('sylius.factory.admin_user')->createNew();

$admin->setEmail('administrator@test.com');
$admin->setPlainPassword('pswd');

$this->container->get('sylius.repository.admin_user')->add($admin);
```

Administration Security

In **Monofony** by default you have got the administration panel routes (`/admin/*`) secured by a firewall - its configuration can be found in the `config/packages/security.yaml` file.

Only the logged in **AdminUsers** are eligible to enter these routes.

Learn more

Note: To learn more, read the UserBundle documentation.

Customer and AppUser

For handling customers of your system AppUser is using a combination of two entities - Customer and AppUser. The difference between these two entities is simple: the Customer is a guest on your application and the AppUser is a user registered in the system - they have an account.

Customer

The Customer entity was created to collect data about non-registered guests of the system - ones that has been buying without having an account or that have somehow provided us their e-mail.

How to create a Customer programmatically?

As usually, use a factory. The only required field for the Customer entity is email, provide it before adding it to the repository.

```
/** @var CustomerInterface $customer */
$customer = $this->container->get('sylius.factory.customer')->createNew();

$customer->setEmail('customer@test.com');

$this->container->get('sylius.repository.customer')->add($customer);
```

The Customer entity can of course hold other information besides an email, it can be for instance `billingAddress` and `shippingAddress`, `firstName`, `lastName` or `birthday`.

Note: The relation between the Customer and AppUser is bidirectional. Both entities hold a reference to each other.

AppUser

AppUser entity is designed for customers that have registered in the system - they have an account with both e-mail and password. They can visit and modify their account.

While creating new account the existence of the provided email in the system is checked - if the email was present - it will already have a Customer therefore the existing one will be assigned to the newly created AppUser, if not - a new Customer will be created together with the AppUser.

How to create an AppUser programmatically?

Assuming that you have a Customer (either retrieved from the repository or a newly created one) - use a factory to create a new AppUser, assign the existing Customer and a password via the setPlainPassword() method.

```
/** @var ShopUserInterface $user */
$user = $this->container->get('sylius.factory.app_user')->createNew();

// Now let's find a Customer by their e-mail:
/** @var CustomerInterface $customer */
$customer = $this->container->get('sylius.repository.customer')->findOneBy(['email' =>
↪ 'customer@test.com']);

// and assign it to the ShopUser
$user->setCustomer($customer);
$user->setPlainPassword('pswd');

$this->container->get('sylius.repository.app_user')->add($user);
```

Changing the AppUser password

The already set password of an AppUser can be easily changed via the setPlainPassword() method.

```
$user->getPassword(); // returns encrypted password - 'pswd'

$user->setPlainPassword('resu1');
// the password will now be 'resu1' and will become encrypted while saving the user␣
↪in the database
```

Customer related events

Event id	Description
`sylius.customer.post_register`	dispatched when a new Customer is registered
`sylius.customer.pre_update`	dispatched when a Customer is updated
`sylius.oauth_user.post_create`	dispatched when an OAuthUser is created
`sylius.oauth_user.post_update`	dispatched when an OAuthUser is updated
`sylius.app_user.post_create`	dispatched when a User is created
`sylius.app_user.post_update`	dispatched when a User is updated
`sylius.app_user.pre_delete`	dispatched before a User is deleted
`sylius.user.email_verification.token`	dispatched when a verification token is requested
`sylius.user.password_reset.request.token`	dispatched when a reset password token is requested
`sylius.user.pre_password_change`	dispatched before user password is changed
`sylius.user.pre_password_reset`	dispatched before user password is reset
`sylius.user.security.implicit_login`	dispatched when an implicit login is done
`security.interactive_login`	dispatched when an interactive login is done

Learn more:

Note: To learn more read: * the SyliusUserBundle documentation. * the SyliusCustomerBundle documentation.

- *AdminUser*
- *Customer and AppUser*
- *AdminUser*
- *Customer and AppUser*
- *Architecture*
- *Fixtures*
- *Users*
- *AdminUser*
- *Customer and AppUser*

CHAPTER 2

The Cookbook

The Cookbook is a collection of specific solutions for specific needs.

2.1 The Cookbook

2.1.1 Entities

How to configure your first resource

As an example we will take an **Article entity**.

```
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;
use Sylius\Component\Resource\Model\ResourceInterface;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 * @ORM\Table(name="app_article")
 */
class Article implements ResourceInterface
{
    use IdentifiableTrait;

    /**
     * @var string|null
     *
     * @ORM\Column(type="string")
     *
     * @Assert\NotBlank()
     */
```

(continues on next page)

(continued from previous page)

```
    private $title;

    /**
     * @return string|null
     */
    public function getTitle(): ?string
    {
        return $this->title;
    }

    /**
     * @param string|null $title
     */
    public function setTitle(?string $title): void
    {
        $this->title = $title;
    }
}
```

If you don't add a form type, it uses a default form type. But it is a good practice to have one.

```
// src/Form/Type/ArticleType.php

namespace App\Form\Type;

use App\Entity\Article;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ArticleType extends AbstractType
{
    /**
    * {@inheritdoc}
    */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        parent::buildForm($builder, $options);
        $builder
            ->add('title', TextType::class, [
                'label' => 'sylius.ui.title',
            ]);
    }

    /**
    * {@inheritdoc}
    */
    public function getBlockPrefix(): string
    {
        return 'app_article';
    }

    /**
     * {@inheritdoc}
     */
    public function configureOptions(OptionsResolver $resolver): void
```

(continues on next page)

(continued from previous page)

```
    {
        $resolver->setDefaults([
            'data_class' => Article::class
        ]);
    }
}
```

You now have to add it on Sylius Resource configuration.

```yaml
# config/sylius/resources.yaml

sylius_resource:
    resources:
        app.article:
            classes:
                model: App\Entity\Article
                form: App\Form\Type\ArticleType
```

Warning: Don't forget to synchronize your database using Doctrine Migrations.

You can use these two commands to generate and synchronize your database.

```
$ bin/console doctrine:migrations:diff
$ bin/console doctrine:migrations:migrate
```

Learn More

- Sylius Resource Bundle documentation
- Doctrine migrations documentation

How to manage your new entity on the admin panel

To add a new grid, create a new grid configuration file in `config/packages/grids/backend/` and import this to sylius_grid configuration file

Create a new grid configuration file

```yaml
# config/sylius/grids/backend/article.yaml

sylius_grid:
    grids:
        app_backend_article:
            driver:
                name: doctrine/orm
                options:
                    class: "%app.model.article.class%"
            sorting:
                title: asc
            fields:
```

(continues on next page)

(continued from previous page)

```
                title:
                    type: string
                    label: sylius.ui.title
                    sortable: null
            filters:
                search:
                    type: string
                    label: sylius.ui.search
                    options:
                        fields: [title]
            actions:
                main:
                    create:
                        type: create
                item:
                    update:
                        type: update
                    delete:
                        type: delete
```

Warning: You need to clear the Symfony cache when creating a new sylius grid configuration file.

Manually importing you sylius_grid configuration (optional)

Grids configuration files are automatically detected when clearing the cache. You can manually add them if you prefer.

```
# config/sylius/grids.yaml

imports:
    - { resource: 'grids/backend/article.yaml' }
    - { resource: 'grids/backend/admin_user.yaml' }
    - { resource: 'grids/backend/customer.yaml' }
```

Learn More

- Configuring grid in sylius documentation
- The whole configuration reference in sylius documentation

How to configure backend routes

To configure backend routes for your entity, you have to create a new file on backend routes folder `config/routes/backend`.

Let's configure our "Article" routes as an example.

```
# config/routes/backend/article.yaml

app_backend_article:
    resource: |
        alias: app.article
```

(continues on next page)

(continued from previous page)

```
        section: backend
        except: ['show']
        redirect: update
        grid: app_backend_article
        vars:
            all:
                subheader: app.ui.manage_articles
            index:
                icon: newspaper
        templates: backend/crud
    type: sylius.resource
```

And add it on backend routes configuration.

```
# config/routes/backend/_main.yaml

[...]

app_backend_article:
    resource: "article.yaml"
```

And that's all!

Learn More

- Configuring routes in sylius documentation

How to configure backend menu

To configure backend menu for your entity, you have to edit *src/Menu/AdminMenuBuilder.php*.

```
// src/Menu/AdminMenuBuilder.php

public function createMenu(array $options): ItemInterface
{
    // add method ...
    $this->addContentSubMenu($menu);
    // rest of the code

    return $menu;
}

/**
 * @param ItemInterface $menu
 *
 * @return ItemInterface
 */
private function addContentSubMenu(ItemInterface $menu): ItemInterface
{
    $content = $menu
        ->addChild('content')
        ->setLabel('sylius.ui.content')
    ;
```

(continues on next page)

(continued from previous page)

```
    $content->addChild('backend_article', ['route' => 'app_backend_article_index'])
        ->setLabel('app.ui.articles')
        ->setLabelAttribute('icon', 'newspaper');

    return $content;
}
```

- *How to configure your first resource*
- *How to manage your new entity on the admin panel*
- *How to configure backend routes*
- *How to configure backend menu*

2.1.2 Fixtures

How to configure a fixture factory

First you have to create a fixture factory. This service is responsible to create new instance of the resource and handle options. This allows to combine random and custom data on your data fixtures.

```
namespace App\Fixture\Factory;

use App\Entity\Article;
use Monofony\Plugin\FixturesPlugin\Fixture\Factory\AbstractExampleFactory;
use Sylius\Component\Resource\Factory\FactoryInterface;
use Symfony\Component\OptionsResolver\Options;
use Symfony\Component\OptionsResolver\OptionsResolver;

final class ArticleExampleFactory extends AbstractExampleFactory
{
    /** @var FactoryInterface */
    private $articleFactory;

    /** @var \Faker\Generator */
    private $faker;

    /** @var OptionsResolver */
    private $optionsResolver;

    public function __construct(FactoryInterface $articleFactory)
    {
        $this->articleFactory = $articleFactory;

        $this->faker = \Faker\Factory::create();
        $this->optionsResolver = new OptionsResolver();

        $this->configureOptions($this->optionsResolver);
    }

    /**
     * {@inheritdoc}
     */
    protected function configureOptions(OptionsResolver $resolver): void
    {
```

(continues on next page)

(continued from previous page)

```
        $resolver
            ->setDefault('title', function (Options $options) {
                return ucfirst($this->faker->words(3, true));
            });
    }

    /**
     * {@inheritdoc}
     */
    public function create(array $options = []): Article
    {
        $options = $this->optionsResolver->resolve($options);

        /** @var Article $article */
        $article = $this->articleFactory->createNew();
        $article->setTitle($options['title']);

        return $article;
    }
}
```

Thanks to services configuration, your new service is already registered and ready to use:

```
$ bin/console debug:container App\Fixture\Factory\ArticleExampleFactory
```

How to configure your fixture options

Now you have to create a fixture service. This defines options you can use on fixtures bundle configurations yaml files.

```
namespace App\Fixture;

use Monofony\Plugin\FixturesPlugin\Fixture\AbstractResourceFixture;
use Symfony\Component\Config\Definition\Builder\ArrayNodeDefinition;

class ArticleFixture extends AbstractResourceFixture
{
    public function __construct(ObjectManager $objectManager, ArticleExampleFactory
↪$articleExampleFactory)
    {
        parent::__construct($objectManager, $articleExampleFactory);
    }

    /**
     * {@inheritdoc}
     */
    public function getName(): string
    {
        return 'article';
    }

    /**
     * {@inheritdoc}
     */
    protected function configureResourceNode(ArrayNodeDefinition $resourceNode)
    {
```

(continues on next page)

(continued from previous page)

```
        $resourceNode
            ->children()
                ->scalarNode('title')->cannotBeEmpty()->end()
        ;
    }
}
```

In this file we have only one custom option which is the article title.

Thanks to autowiring system, you can already use it.

```
$ bin/console debug:container App\Fixture\ArticleFixture
```

How to use it on your suite

```
# config/sylius/fixtures.yaml

sylius_fixtures:
    suites:
        default:
            listeners:
                orm_purger: ~
                logger: ~

            fixtures:
                [...]

                article:
                    options:
                        random: 10
                        custom:
                            -
                                title: "Awesome article"
```

it will generates 10 random articles and one custom with title `Awesome article`.

How to load your data fixtures

You can load all your data fixtures using this command.

```
$ bin/console sylius:fixtures:load
```

- *How to configure a fixture factory*
- *How to configure your fixture options*
- *How to use it on your suite*
- *How to load your data fixtures*

2.1.3 BDD

How to use Phpspec to design your code

How to configure phpspec with code coverage

By default, phpspec on Monofony is configured with code coverage which needs xdebug or phpdbg installed. Thus you have two options: * install xdebug * install phpdbg (easier and faster)

Note: But if you don't need that feature, *disable code coverage*.

Install phpdbg

```
$ # on linux
$ sudo apt-get install php7.2-phpdbg
$
$ # on max
$ brew install php72-phpdbg
```

How to disable phpspec code coverage

```
$ cp phpspec.yml.dist phpspec
```

And just comment the content

```
# extensions:
#    LeanPHP\PhpSpec\CodeCoverage\CodeCoverageExtension: ~
```

How to design entities with phpspec

Lets configure an Article entity with a title and an author. Title is a simple string and author implements CustomerInterface.

Warning: By default, phpspec on Monofony is configured with code coverage. *Learn how to configure phpspec with code coverage* or *disable code coverage*.

Generate phpspec for your entity

```
$ vendor/bin/phpspec describe App/Entity/Article

$ # with phpdbg installed
$ phpdbg -qrr vendor/bin/phpspec describe App/Entity/Article
```

```
# spec/src/App/Entity/Article.php

namespace spec\App\Entity;

use App\Entity\Article;
use PhpSpec\ObjectBehavior;
```

(continues on next page)

(continued from previous page)

```
use Prophecy\Argument;

class ArticleSpec extends ObjectBehavior
{
    function it_is_initializable()
    {
        $this->shouldHaveType(Article::class);
    }
}
```

Run phpspec and do not fear Red

To run phpspec for our Article entity, run this command:

```
$ vendor/bin/phpspec run spec/App/Entity/ArticleSpec.php -n
$
$ # with phpdbg installed
$ phpdbg -qrr vendor/bin/phpspec run spec/App/Entity/ArticleSpec.php -n
```

And be happy with your first error message with red color.

Note: You can simply run all the phpspec tests by running *vendor/bin/phpspec run -n*

Create a minimal Article class

```
# src/App/Entity/Article.php

namespace App\Entity;

class Article
{
}
```

Rerun phpspec and see a beautiful green color.

Specify it implements sylius resource interface

```
function it_implements_sylius_resource_interface(): void
{
    $this->shouldImplement(ResourceInterface::class);
}
```

Warning: And Rerun phpspec, DO NOT FEAR RED COLOR! It's important to check that you write code which solves your specifications.

Solve this on your entity

```
# src/App/Entity/Article.php

namespace App\Entity;

use Sylius\Component\Resource\Model\ResourceInterface;

class Article implements ResourceInterface
{
    use IdentifiableTrait;
}
```

Warning: Rerun phpspec again and check this specification is solved.

Specify title behaviours

```
function it_has_no_title_by_default(): void
{
    $this->getTitle()->shouldReturn(null);
}

function its_title_is_mutable(): void
{
    $this->setTitle('This documentation is so great');
    $this->getTitle()->shouldReturn('This documentation is so great');
}
```

Warning: Don't forget to rerun phpspec on each step.

Add title on Article entity

```
# src/App/Entity/Article.php

/**
 * @var string|null
 */
private $title;

/**
 * @return string|null
 */
public function getTitle(): ?string
{
    return $this->title;
}

/**
 * @param string|null $title
```

(continues on next page)

(continued from previous page)

```
     */
    public function setTitle(?string $title): void
    {
        $this->title = $title;
    }
```

Specify author of the article

```
# spec/src/App/Entity/Article.php

use Sylius\Component\Customer\Model\CustomerInterface;

// [...]

function its_author_is_mutable(CustomerInterface $author): void
{
    $this->setAuthor($author);
    $this->getAuthor()->shouldReturn($author);
}
```

Add author on your entity

```
# src/App/Entity/Article.php

// [...]

/**
 * @var CustomerInterface|null
 */
private $author;

// [...]

/**
 * @return CustomerInterface|null
 */
public function getAuthor(): ?CustomerInterface
{
    return $this->author;
}

/**
 * @param CustomerInterface|null $author
 */
public function setAuthor(?CustomerInterface $author): void
{
    $this->author = $author;
}
```

That's all to design your first entity!

How to design services with phpspec

Lets configure an Article factory to create an article for an author. This Author implements CustomerInterface.

Generate phpspec for your entity factory

```
$ vendor/bin/phpspec describe App/Factory/ArticleFactory

$ # with phpdbg installed
$ phpdbg -qrr vendor/bin/phpspec describe App/Factory/ArticleFactory
```

Run phpspec and do not fear Red

To run phpspec for our Article factory, run this command:

```
$ vendor/bin/phpspec run spec/App/Factory/ArticleFactory.php -n
$
$ # with phpdbg installed
$ phpdbg -qrr vendor/bin/phpspec run spec/App/Factory/ArticleFactorySpec.php -n
```

And be happy with your first error message with red color.

Note: You can simply run all the phpspec tests by running *vendor/bin/phpspec run -n*

Create a minimal article factory class

```
# src/Factory/ArticleFactory.php

namespace App\Factory;

class ArticleFactory
{
}
```

Rerun phpspec and see a beautiful green color.

Specify it implements sylius factory interface

```
# spec/App/Factory/ArticleFactorySpec.php

function it_implements_sylius_factory_interface(): void
{
    $this->shouldImplement(FactoryInterface::class);
}
```

Warning: Don't forget to rerun phpspec on each step.

Solve this on your factory

```
# src/Factory/ArticleFactory.php

namespace App\Factory;

use Sylius\Component\Resource\Factory\FactoryInterface;

class ArticleFactory implements FactoryInterface
{
    /**
     * {@inheritdoc}
     */
    public function createNew()
    {
    }
}
```

Specify it creates articles

```
# spec/App/Factory/ArticleFactorySpec.php

// [...]

function its_creates_articles(): void
{
    $article = $this->createNew();

    $article->shouldImplement(Article::class);
}
```

Solve this on your factory

```
# src/Factory/ArticleFactory.php

namespace App\Factory;

use Sylius\Component\Resource\Factory\FactoryInterface;

class ArticleFactory implements FactoryInterface
{
    /** @var string */
    private $className;

    public function __construct(string $className)
    {
        $this->className = $className;
    }

    /**
     * {@inheritdoc}
     */
    public function createNew(): Article
```

(continues on next page)

(continued from previous page)

```
    {
        return new $this->className();
    }
}
```

Running this step will throw this exception:

```
exception [err:ArgumentCountError("Too few arguments to function␣
↪App\Factory\ArticleFactory::__construct(), 0 passed and exactly 1 expected")] has␣
↪been thrown.
```

To add arguments on constructor, go back to your factory spec and add these lines:

```
# spec/App/Factory/ArticleFactorySpec.php

namespace spec\App\Factory;

use App\Entity\Article;
use App\Factory\ArticleFactory;
use PhpSpec\ObjectBehavior;
use Sylius\Component\Resource\Factory\FactoryInterface;

class ArticleFactorySpec extends ObjectBehavior
{
    function let()
    {
        $this->beConstructedWith(Article::class);
    }

    // [...]
}
```

Rerun phpspec and it should be solved.

Note: Here you pass a string, but you often need to pass objects on constructor. You just have to add them on arguments of the let method and don't forget to use typehints.

Here is an example with object arguments:

```
function let(FactoryInterface $factory)
{
    $this->beConstructedWith($factory);
}
```

Specify it creates articles for an author

```
# spec/App/Factory/ArticleFactorySpec.php

// [...]

function its_creates_articles_for_an_author(CustomerInterface $author): void
{
    $article = $this->createForAuthor($author);
```

(continues on next page)

(continued from previous page)

```
    $article->getAuthor()->shouldReturn($author);
}
```

Add this method on your factory

```
# src/Factory/ArticleFactory.php

// [...]

/**
 * @param CustomerInterface $author
 *
 * @return Article
 */
public function createForAuthor(CustomerInterface $author): Article
{
    $article = $this->createNew();
    $article->setAuthor($author);

    return $article;
}
```

And that's all to specify this simple article factory.

- *How to configure phpspec with code coverage*
- *How to disable phpspec code coverage*
- *How to design entities with phpspec*
- *How to design services with phpspec*

Learn more

Note: To learn more, read the Phpspec documentation.

How to use Behat to design your features

Note: This section is based on the great Sylius documentation.

Behaviour driven development is an approach to software development process that provides software development and management teams with shared tools and a shared process to collaborate on software development. The awesome part of BDD is its ubiquitous language, which is used to describe the software in English-like sentences of domain specific language.

The application's behaviour is described by scenarios, and those scenarios are turned into automated test suites with tools such as Behat.

Sylius behaviours are fully covered with Behat scenarios. There are more than 1200 scenarios in the Sylius suite, and if you want to understand some aspects of Sylius better, or are wondering how to configure something, we strongly recommend reading them. They can be found in the `features/` directory of the Sylius/Sylius repository.

We use FriendsOfBehat/SymfonyExtension to integrate Behat with Symfony.

Basic Usage

The best way of understanding how things work in detail is showing and analyzing examples, that is why this section gathers all the knowledge from the previous chapters. Let's assume that we are going to implement the functionality of managing countries in our system. Now let us show you the flow.

Describing features

Let's start with writing our feature file, which will contain answers to the most important questions: Why (benefit, business value), who (actor using the feature) and what (the feature itself). It should also include scenarios, which serve as examples of how things supposed to work. Let's have a look at the `features/addressing/managing_countries/adding_country.feature` file.

```
# features/addressing/managing_countries/adding_country.feature

@managing_countries
Feature: Adding a new country
    In order to sell my goods to different countries
    As an Administrator
    I want to add a new country to the store

    Background:
        Given I am logged in as an administrator

    @ui
    Scenario: Adding country
        When I want to add a new country
        And I choose "United States"
        And I add it
        Then I should be notified that it has been successfully created
        And the country "United States" should appear in the store
```

Pay attention to the form of these sentences. From the developer point of view they are hiding the details of the feature's implementation. Instead of describing "When I click on the select box And I choose United States from the dropdown Then I should see the United States country in the table" - we are using sentences that are less connected with the implementation, but more focused on the effects of our actions. A side effect of such approach is that it results in steps being really generic, therefore if we want to add another way of testing this feature for instance in the domain or api context, it will be extremely easy to apply. We just need to add a different tag (in this case "@domain") and of course implement the proper steps in the domain context of our system. To be more descriptive let's imagine that we want to check if a country is added properly in two ways. First we are checking if the adding works via frontend, so we are implementing steps that are clicking, opening pages, filling fields on forms and similar, but also we want to check this action regardlessly of the frontend, for that we need the domain, which allows us to perform actions only on objects.

Choosing a correct suite

After we are done with a feature file, we have to create a new suite for it. At the beginning we have decided that it will be a frontend/user interface feature, that is why we are placing it in "config/suites/ui/addressing/managing_countries.yaml".

```
# config/suites/ui/addressing/managing_countries.yaml

default:
    suites:
        ui_managing_countries:
            contexts:
                # This service is responsible for clearing database before each␣
↪scenario,
                # so that only data from the current and its background is available.
                - App\Tests\Behat\Context\Hook\DoctrineORMContext

                # The transformer contexts services are responsible for all the␣
↪transformations of data in steps:
                # For instance "And the country "France" should appear in the store"␣
↪transforms "(the country "France")" to a proper Country object, which is from now␣
↪on available in the scope of the step.
                - App\Tests\Behat\Context\Transform\CountryContext
                - App\Tests\Behat\Context\Transform\SharedStorageContext

                # The setup contexts here are preparing the background, adding␣
↪available countries and users or administrators.
                # These contexts have steps like "I am logged in as an administrator"␣
↪already implemented.
                - App\Tests\Behat\Context\Setup\GeographicalContext
                - App\Tests\Behat\Context\Setup\SecurityContext

                # Lights, Camera, Action!
                # Those contexts are essential here we are placing all action steps␣
↪like "When I choose "France" and I add it Then I should ne notified that...".
                - App\Tests\Behat\Context\Ui\Backend\ManagingCountriesContext
                - App\Tests\Behat\Context\Ui\Backend\NotificationContext
            filters:
                tags: "@managing_countries && @ui"
```

A very important thing that is done here is the configuration of tags, from now on Behat will be searching for all your features tagged with `@managing_countries` and your scenarios tagged with `@ui`.

We have mentioned with the generic steps we can easily switch our testing context to @domain. Have a look how it looks:

```
# config/suites/domain/addressing/managing_countries.yaml

default:
    suites:
        domain_managing_countries:
            contexts:
                - App\Tests\Behat\Context\Hook\DoctrineORMContext

                - App\Tests\Behat\Context\Transform\CountryContext
                - App\Tests\Behat\Context\Transform\SharedStorageContext
```

(continues on next page)

(continued from previous page)

```
                - App\Tests\Behat\Context\Setup\GeographicalContext
                - App\Tests\Behat\Context\Setup\SecurityContext

                # Domain step implementation.
                - App\Tests\Behat\Context\Domain\Backend\ManagingCountriesContext
            filters:
                tags: "@managing_countries && @domain"
```

We are almost finished with the suite configuration.

Registering Pages

The page object approach allows us to hide all the detailed interaction with ui (html, javascript, css) inside.

We have three kinds of pages:

- Page - First layer of our pages it knows how to interact with DOM objects. It has a method `getUrl(array $urlParameters)` where you can define a raw url to open it.
- SymfonyPage - This page extends the Page. It has a router injected so that the `getUrl()` method generates a url from the route name which it gets from the `getRouteName()` method.
- Base Crud Pages (IndexPage, CreatePage, UpdatePage) - These pages extend SymfonyPage and they are specific to the Sylius resources. They have a resource name injected and therefore they know about the route name.

There are two ways to manipulate UI - by using `getDocument()` or `getElement('your_element')`. First method will return a `DocumentElement` which represents an html structure of the currently opened page, second one is a bit more tricky because it uses the `->getDefinedElements(): array` method and it will return a `NodeElement` which represents only the restricted html structure.

Usage example of `getElement('your_element')` and `getDefinedElements()` methods.

```
final class CreatePage extends SymfonyPage implements CreatePageInterface
{
    // This method returns a simple associative array, where the key is the name of␣
↪your element and the value is its locator.
    protected function getDefinedElements(): array
    {
        return array_merge(parent::getDefinedElements(): array, [
            'provinces' => '#sylius_country_provinces',
        ]);
    }

    // By default it will assume that your locator is css.
    // Example with xpath.
    protected function getDefinedElements(): array
    {
        return array_merge(parent::getDefinedElements(): array, [
            'provinces_css' => '.provinces',
            'provinces_xpath' => ['xpath' => '//*[contains(@class, "provinces")]'], //
↪ Now your value is an array where key is your locator type.
        ]);
    }

    // Like that you can easily manipulate your page elements.
    public function addProvince(ProvinceInterface $province): void
```

(continues on next page)

(continued from previous page)

```
    {
        $provinceSelectBox = $this->getElement('provinces');

        $provinceSelectBox->selectOption($province->getName());
    }
}
```

Let's get back to our main example and analyze our scenario. We have steps like:

```
When I choose "France"
And I add it
Then I should be notified that it has been successfully created
And the country "France" should appear in the store
```

```
namespace App\Tests\Behat\Page\Backend\Country;

use App\Tests\Behat\Page\Backend\Crud\CreatePage as BaseCreatePage;

final class CreatePage extends BaseCreatePage implements CreatePageInterface
{
    public function chooseName(string $name): void
    {
        $this->getDocument()->selectFieldOption('Name', $name);
    }

    public function create(): void
    {
        $this->getDocument()->pressButton('Create');
    }
}
```

```
namespace App\Tests\Behat\Page\Backend\Country;

use App\Tests\Behat\Page\Backend\Crud\IndexPage as BaseIndexPage;

final class IndexPage extends BaseIndexPage implements IndexPageInterface
{
    public function isSingleResourceOnPage(array $parameters): bool
    {
        try {
            // Table accessor is a helper service which is responsible for all html␣
↪table operations.
            $rows = $this->tableAccessor->getRowsWithFields($this->getElement('table
↪'), $parameters);

            return 1 === count($rows);
        } catch (ElementNotFoundException $exception) {
            // Table accessor throws this exception when cannot find table element on␣
↪page.
            return false;
        }
    }
}
```

Warning: There is one small gap in this concept - PageObjects is not a concrete instance of the currently opened page, they only mimic its behaviour (dummy pages). This gap will be more understandable on the below code example.

```
// Of course this is only to illustrate this gap.

class HomePage
{
    // In this context on home page sidebar you have for example weather information␣
↪in selected countries.
    public function readWeather()
    {
        return $this->getElement('sidebar')->getText();
    }

    protected function getDefinedElements(): array
    {
        return ['sidebar' => ['css' => '.sidebar']]
    }

    protected function getUrl()
    {
        return 'http://your_domain.com';
    }
}

class LeagueIndexPage
{
    // In this context you have for example football match results.
    public function readMatchResults()
    {
        return $this->getElement('sidebar')->getText();
    }

    protected function getDefinedElements(): array
    {
        return ['sidebar' => ['css' => '.sidebar']]
    }

    protected function getUrl()
    {
        return 'http://your_domain.com/leagues/'
    }
}

final class GapContext implements Context
{
    private $homePage;
    private $leagueIndexPage;

    /**
     * @Given I want to be on Homepage
     */
    public function iWantToBeOnHomePage() // After this method call we will be on
↪"http://your_domain.com".
    {
```

(continues on next page)

(continued from previous page)

```
        $this->homePage->open(); //When we add @javascript tag we can actually see␣
↪this thanks to selenium.
    }

    /**
     * @Then I want to see the sidebar and get information about the weather in France
     */
    public function iWantToReadSideBarOnHomePage($someInformation) // Still "http://
↪your_domain.com".
    {
        $someInformation === $this->leagueIndexPage->readMatchResults() // This␣
↪returns true, but wait a second we are on home page (dummy pages).

        $someInformation === $this->homePage->readWeather() // This also returns true.
    }
}
```

Registering contexts

As it was shown in the previous section we have registered a lot of contexts, so we will show you only some of the steps implementation.

```
Given I want to add a new country
And I choose "United States"
And I add it
Then I should be notified that it has been successfully created
And the country "United States" should appear in the store
```

Let's start with essential one ManagingCountriesContext

Ui contexts

```
namespace App\Tests\Behat\Context\Ui\Backend

use Behat\Behat\Context\Context;

final class ManagingCountriesContext implements Context
{
    /** @var IndexPageInterface */
    private $indexPage;

    /** @var CreatePageInterface */
    private $createPage;

    /** @var UpdatePageInterface */
    private $updatePage;

    public function __construct(
        IndexPageInterface $indexPage,
        CreatePageInterface $createPage,
        UpdatePageInterface $updatePage
    ) {
        $this->indexPage = $indexPage;
```

(continues on next page)

(continued from previous page)

```
        $this->createPage = $createPage;
        $this->updatePage = $updatePage;
    }

    /**
     * @Given I want to add a new country
     */
    public function iWantToAddNewCountry(): void
    {
        $this->createPage->open(); // This method will send request.
    }

    /**
     * @When I choose :countryName
     */
    public function iChoose($countryName): void
    {
        $this->createPage->chooseName($countryName);
        // Great benefit of using page objects is that we hide html manipulation␣
↪behind a interfaces so we can inject different CreatePage which implements␣
↪CreatePageInterface
        // And have different html elements which allows for example chooseName(
↪$countryName).
    }

    /**
     * @When I add it
     */
    public function iAddIt(): void
    {
        $this->createPage->create();
    }

    /**
     * @Then /^the (country "([^"]+)") should appear in the store$/
     */
    public function countryShouldAppearInTheStore(CountryInterface $country): void //␣
↪This step use Country transformer to get Country object.
    {
        $this->indexPage->open();

        //Webmozart assert library.
        Assert::true(
            $this->indexPage->isSingleResourceOnPage(['code' => $country->getCode()]),
            sprintf('Country %s should exist but it does not', $country->getCode())
        );
    }
}
```

```
namespace App\Tests\Behat\Context\Ui\Backend

use Behat\Behat\Context\Context;

final class NotificationContext implements Context
{
    /**
```

(continues on next page)

(continued from previous page)

```
     * This is a helper service which give access to proper notification elements.
     *
     * @var NotificationCheckerInterface
     */
    private $notificationChecker;

    /**
     * @param NotificationCheckerInterface $notificationChecker
     */
    public function __construct(NotificationCheckerInterface $notificationChecker)
    {
        $this->notificationChecker = $notificationChecker;
    }

    /**
     * @Then I should be notified that it has been successfully created
     */
    public function iShouldBeNotifiedItHasBeenSuccessfullyCreated(): void
    {
        $this->notificationChecker->checkNotification('has been successfully created.
↪', NotificationType::success());
    }
}
```

Transformer contexts

```
namespace App\Tests\Behat\Context\Transform;

use Behat\Behat\Context\Context;

final class CountryContext implements Context
{
    /** @var CountryNameConverterInterface */
    private $countryNameConverter;

    /** @var RepositoryInterface */
    private $countryRepository;

    public function __construct(
        CountryNameConverterInterface $countryNameConverter,
        RepositoryInterface $countryRepository
    ) {
        $this->countryNameConverter = $countryNameConverter;
        $this->countryRepository = $countryRepository;
    }

    /**
     * @Transform /^country "([^"]+)"$/
     * @Transform /^"([^"]+)" country$/
     */
    public function getCountryByName(string $countryName): Country // Thanks to this␣
↪method we got in our ManagingCountries an Country object.
    {
        $countryCode = $this->countryNameConverter->convertToCode($countryName);
```

(continues on next page)

(continued from previous page)

```
        $country = $this->countryRepository->findOneBy(['code' => $countryCode]);

        Assert::notNull(
            $country,
            'Country with name %s does not exist'
        );

        return $country;
    }
}
```

```
namespace App\Tests\Behat\Context\Ui\Backend;

use App\Tests\Behat\Page\Backend\Country\UpdatePageInterface;
use Behat\Behat\Context\Context;

final class ManagingCountriesContext implements Context
{
    /** @var UpdatePageInterface */
    private $updatePage;

    public function __construct(UpdatePageInterface $updatePage)
    {
        $this->updatePage = $updatePage;
    }

    /**
     * @Given /^I want to create a new province in (country "[^"]+")$/
     */
    public function iWantToCreateANewProvinceInCountry(CountryInterface $country)
    {
        $this->updatePage->open(['id' => $country->getId()]);

        $this->updatePage->clickAddProvinceButton();
    }
}
```

```
namespace App\Tests\Behat\Context\Transform;

use Behat\Behat\Context\Context;

final class ShippingMethodContext implements Context
{
    /** @var ShippingMethodRepositoryInterface */
    private $shippingMethodRepository;

    public function __construct(ShippingMethodRepositoryInterface
↪$shippingMethodRepository)
    {
        $this->shippingMethodRepository = $shippingMethodRepository;
    }

    /**
     * @Transform :shippingMethod
     */
    public function getShippingMethodByName($shippingMethodName)
```

(continues on next page)

(continued from previous page)

```
    {
        $shippingMethod = $this->shippingMethodRepository->findOneByName(
↪$shippingMethodName);
        if (null === $shippingMethod) {
            throw new \Exception('Shipping method with name "'.$shippingMethodName.'"␣
↪does not exist');
        }

        return $shippingMethod;
    }
}
```

```
namespace App\Tests\Behat\Context\Ui\Admin;

use App\Tests\Behat\Page\Admin\ShippingMethod\UpdatePageInterface;
use Behat\Behat\Context\Context;

final class ShippingMethodContext implements Context
{
    /** @var UpdatePageInterface */
    private $updatePage;

    public function __construct(UpdatePageInterface $updatePage)
    {
        $this->updatePage = $updatePage;
    }

    /**
     * @Given I want to modify a shipping method :shippingMethod
     */
    public function iWantToModifyAShippingMethod(ShippingMethodInterface
↪$shippingMethod)
    {
        $this->updatePage->open(['id' => $shippingMethod->getId()]);
    }
}
```

Warning: Contexts should have single responsibility and this segregation (Setup, Transformer, Ui, etc…) is not accidental. We shouldn't create objects in transformer contexts.

Setup contexts

For setup context we need different scenario with more background steps and all preparing scene steps. Editing scenario will be great for this example:

Scenario:

```
Given the store has disabled country "France"
And I want to edit this country
When I enable it
And I save my changes
Then I should be notified that it has been successfully edited
And this country should be enabled
```

```php
namespace App\Tests\Behat\Context\Setup;

use Behat\Behat\Context\Context;

final class GeographicalContext implements Context
{
    /** @var SharedStorageInterface */
    private $sharedStorage;

    /** @var FactoryInterface */
    private $countryFactory;

    /** @var RepositoryInterface */
    private $countryRepository;

    /** @var CountryNameConverterInterface */
    private $countryNameConverter;

    public function __construct(
        SharedStorageInterface $sharedStorage,
        FactoryInterface $countryFactory,
        RepositoryInterface $countryRepository,
        CountryNameConverterInterface $countryNameConverter
    ) {
        $this->sharedStorage = $sharedStorage;
        $this->countryFactory = $countryFactory;
        $this->countryRepository = $countryRepository;
        $this->countryNameConverter = $countryNameConverter;
    }

    /**
     * @Given /^the store has disabled country "([^"]*)"$/
     */
    public function theStoreHasDisabledCountry($countryName) // This method save␣
↪country in data base.
    {
        $country = $this->createCountryNamed(trim($countryName));
        $country->disable();

        $this->sharedStorage->set('country', $country);
        // Shared storage is an helper service for transferring objects between steps.
        // There is also SharedStorageContext which use this helper service to␣
↪transform sentences like "(this country), (it), (its), (theirs)" into Country␣
↪Object.

        $this->countryRepository->add($country);
    }

    private function createCountryNamed(string $name): CountryInterface
    {
        /** @var CountryInterface $country */
        $country = $this->countryFactory->createNew();
        $country->setCode($this->countryNameConverter->convertToCode($name));

        return $country;
    }
}
```

How to add a new context?

Thanks to symfony autowiring, most of your contexts are ready to use.

But if you need to manually route an argument, it is needed to add a service in `config/services_test.yaml` file.

```yaml
App\Tests\Behat\Context\CONTEXT_CATEGORY\CONTEXT_NAME:
    arguments:
        $specificArgument: App\SpecificArgument
```

Then you can use it in your suite configuration:

```yaml
default:
    suites:
        SUITE_NAME:
            contexts:
                - 'App\Tests\Behat\Context\CONTEXT_CATEGORY\CONTEXT_NAME'
            filters:
                tags: "@SUITE_TAG"
```

Note: The context categories are usually one of `cli`, `hook`, `setup`, `transform`, `ui`.

How to add a new page object?

Sylius uses a solution inspired by `SensioLabs/PageObjectExtension`, which provides an infrastructure to create pages that encapsulates all the user interface manipulation in page objects.

To create a new page object it is needed to add a service.

The simplest Symfony-based page looks like:

```php
use FriendsOfBehat\PageObjectExtension\Page\SymfonyPage;

class LoginPage extends SymfonyPage
{
    public function getRouteName(): string
    {
        return 'app_frontend_security_login';
    }
}
```

Note: There are some boilerplates for common pages, which you may use. The available parents are `FriendsOfBehat\PageObjectExtension\Page\Page` and `FriendsOfBehat\PageObjectExtension\Page\SymfonyPage`. It is not required for a page to extend any class as pages are POPOs (Plain Old PHP Objects).

How to define a new suite?

To define a new suite it is needed to create a suite configuration file in a one of `cli`/`ui` directory inside `config/suites`. Then register that file in `config/suites.yaml`.

How to use transformers?

Behat provides many awesome features, and one of them is definitely **transformers**. They can be used to transform (usually widely used) parts of steps and return some values from them, to prevent unnecessary duplication in many steps' definitions.

Basic transformer

Example is always the best way to clarify, so let's look at this:

```
/**
 * @Transform /^"([^"]+)" shipping method$/
 * @Transform /^shipping method "([^"]+)"$/
 * @Transform :shippingMethod
 */
public function getShippingMethodByName($shippingMethodName)
{
    $shippingMethod = $this->shippingMethodRepository->findOneByName(
↪$shippingMethodName);

    Assert::notNull(
        $shippingMethod,
        sprintf('Shipping method with name "%s" does not exist', $shippingMethodName)
    );

    return $shippingMethod;
}
```

This transformer is used to return `ShippingMethod` object from proper repository using it's name. It also throws exception if such a method does not exist. It can be used in plenty of steps, that have shipping method name in it.

Note: In the example above a Webmozart assertion library was used, to assert a value and throw an exception if needed.

But how to use it? It is as simple as that:

```
/**
 * @Given /^(shipping method "[^"]+") belongs to ("[^"]+" tax category)$/
 */
public function shippingMethodBelongsToTaxCategory(
    ShippingMethodInterface $shippingMethod,
    TaxCategoryInterface $taxCategory
) {
    // some logic here
}
```

If part of step matches transformer definition, it should be surrounded by parenthesis to be handled as whole expression. That's it! As it is shown in the example, many transformers can be used in the same step definition. Is it all? No! The following example will also work like charm:

```
/**
 * @When I delete shipping method :shippingMethod
 * @When I try to delete shipping method :shippingMethod
 */
```

(continues on next page)

(continued from previous page)

```
public function iDeleteShippingMethod(ShippingMethodInterface $shippingMethod)
{
    // some logic here
}
```

It is worth to mention, that in such a case, transformer would be matched depending on a name after ':' sign. So many transformes could be used when using this signature also. This style gives an opportunity to write simple steps with transformers, without any regex, which would boost context readability.

Note: Transformer definition does not have to be implemented in the same context, where it is used. It allows to share them between many different contexts.

Transformers implemented in Sylius

Specified

There are plenty of transformers already implemented in *Sylius*. Most of them return specific resources from their repository, for example:

- `tax category "Fruits"` -> find tax category in their repository with name "Fruits"
- `"Chinese banana" variant of product "Banana"` -> find variant of specific product

etc. You're free to use them in your own behat scenarios.

Note: All transformers definitions are currently kept in `App\Tests\Behat\Context\Transform` namespace.

Warning: Remember to include contexts with transformers in custom suite to be able to use them!

Generic

Moreover, there are also some more generic transformers, that could be useful in many different cases. They are now placed in two contexts: `LexicalContext` and `SharedStorageContext`. Why are they so awesome? Let's describe them one by one:

LexicalContext

- `@Transform /^"(?:€|£|\$)((?:\d+\.)?\d+)"$/` -> tricky transformer used to parse price string with currency into integer (used to represent price in *Sylius*). It is used in steps like `this promotion gives "€30.00" fixed discount to every order`
- `@Transform /^"((?:\d+\.)?\d+)%"$/` -> similar one, transforming percentage string into float (example: `this promotion gives "10%" percentage discount to every order`)

SharedStorageContext

Note: `SharedStorage` is kind of container used to keep objects, which can be shared between steps. It can be used, for example, to keep newly created promotion, to use its name in checking existence step.

- `@Transform /^(it|its|theirs)$/` -> amazingly useful transformer, that returns last resource saved in `SharedStorage`. It allows to simplify many steps used after creation/update (and so on) actions. Example: instead of writing `When I create "Wade Wilson" customer/Then customer "Wade Wilson" should be registered` just write `When I create "Wade Wilson" customer/ Then it should be registered`
- `@Transform /^(?:this|that|the) ([^"]+)$/` -> similar to previous one, but returns resource saved with specific key, for example `this promotion` will return resource saved with `promotion` key in `SharedStorage`

How to change Behat application base url

By default Behat uses `https://localhost:8080/` as your application base url. If your one is different, you need to create `behat.yml` files that will overwrite it with your custom url:

```
# behat.yml

imports: ["behat.yml.dist"]

default:
    extensions:
        Behat\MinkExtension:
            base_url: http://my.custom.url
```


Learn more

Note: To learn more, read the Behat documentation.

2.1.4 Dashboard

How to create your own statistics

The mechanism behind the displaying of statistics relies on tagged services which are supported since Symfony 4.3

Create your own statistic

Add a new class to `/src/Dashboard/Statistic` and make sure it implement the `App\Dashboard\Statistics\StatisticInterface`. This way it will be automatically tagged with `app.dashboard_statistic` which is used to fetch all existing statistics.

It also enforces you to implement a function called `generate()` which need to return a string.

Note: The response of the generate function will be displayed as is in the dashboard. Which means you can return anything, as long as it is a string. Eg. in the CustomerStatistic it is an HTML block which shows you the amount of registered customers.

Order your statistics

Since Symfony 4.4 it is possible to sort your services with a static function called `getDefaultPriority`. Here you need to return an integer to set the weight of the service. Statistics with a higher priority will be displayed first. This is why we chose to work with negative values. (-1 for the first element, -2 for the second,...). But feel free to use your own sequence when adding more statistics.

```
public static function getDefaultPriority(): int
{
    return -1;
}
```

Warning: If you change the priority it is necessary to clear your cache. Otherwise you won't see the difference.

Add custom logic to your statistic

Because all statistics are services it's perfectly possible to do anything with them as long as the generate function returns a string. So you can inject any service you want trough Dependency Injection to build your statistic.

Basic example

This is a basic example. It fetches and renders the amount of registered customers.

```
namespace App\Dashboard\Statistics;

use App\Repository\CustomerRepository;
use Monofony\Component\Admin\Dashboard\Statistics\StatisticInterface;
use Symfony\Component\Templating\EngineInterface;

class CustomerStatistic implements StatisticInterface
{
    /** @var CustomerRepository */
    private $customerRepository;

    /** @var EngineInterface */
    private $engine;
```

(continues on next page)

(continued from previous page)

```
    public function __construct(CustomerRepository $customerRepository,␣
↪EngineInterface $engine)
    {
        $this->customerRepository = $customerRepository;
        $this->engine = $engine;
    }

    public function generate(): string
    {
        $amountCustomers = $this->customerRepository->countCustomers();

        return $this->engine->render('backend/dashboard/statistics/_amount_of_
↪customers.html.twig', [
            'amountOfCustomers' => $amountCustomers,
        ]);
    }

    public static function getDefaultPriority(): int
    {
        return -1;
    }
}
```

- *How to create your own statistics*
- *Basic example*

CHAPTER 3

Deployment

The Deployment guide helps to deploy the website on servers.

3.1 Deployment

3.1.1 Authorized keys API

Adding ssh authorized keys for server on your local computer

```
$ cat ~/.ssh/id_rsa.pub | ssh mobizel@XXX.XXX.XX.XX "cat - >> ~/.ssh/authorized_keys"
```

and enter the correct password for username “mobizel” on server

3.1.2 Deploy the staging environment

```
$ bundle exec "cap staging deploy"
```

3.1.3 Deploy the production environment

```
$ bundle exec "cap production deploy"
```

- *Deployment*

Index