Behat Magento 2 Extension

Apr 04, 2020

Contents

1 Guide **3**
- 1.1 Quick start . . . 3
- 1.2 Installation . . . 5
- 1.3 Configuration . . . 6
- 1.4 Usage Examples . . . 11

2 References **21**

This Behat extension provides provides a custom service container for Behat which allows to inject Magento services into Behat Contexts and Behat helper services.

CHAPTER 1

Guide

1.1 Quick start

1.1.1 Install Behat

If you didn't install Behat already, then you can install it with composer in the following way:

```
$ composer require --dev behat/behat
```

For alternative installation options check the Behat official documentation

1.1.2 Install the Extension

Similarly you can install the extension via composer:

```
$ composer require --dev bex/behat-magento2-extension
```

For more information see the the *installation section of this documentation*.

1.1.3 Setup the Behat configuration

You need to enable the extension in the Behat configuration and configure your Behat Suite to use the Magento 2 Service Container. Your `behat.yml` should look like this:

```
default:
  extensions:
    Bex\Behat\Magento2Extension: ~

  suites:
    application:
      autowire: true
```

(continues on next page)

(continued from previous page)

```
    contexts:
      - FeatureContext

    services: '@bex.magento2_extension.service_container'
```

With the above configuration:

- The extension is enabled
- The `application` suite uses the Behat Magento 2 service container
- The Behat Context dependencies are autowired

For more detailed information see the *configuration section of this documentation*.

1.1.4 Verify the configuration

In order to verify that the extension is configured correctly you will need a test feature. For example create a `features/my_feature.feature` file like this:

```
Feature: Magento and Behat DI connected
  As a developer
  In order to write Behat tests easily
  I should be able to inject services from the Magento DI into Behat Contexts

  Scenario: Injecting service from Magento DI to Behat Context as argument␣
↪for Behat Context constructor
    Given A service has been successfully injected through the Context␣
↪constructor
    When I work with Behat
    Then I am happy
```

Also to implement the above feature you need to add the following step definitions to your `features/bootstrap/FeatureContext.php` Behat Context:

```
<?php

use Behat\Behat\Context\Context;
use Exception;
use Magento\Sales\Api\OrderRepositoryInterface;

class FeatureContext implements Context
{
    /** @var OrderRepositoryInterface */
    private $orderRepository;

    public function __construct(OrderRepositoryInterface $orderRepository)
    {
        $this->orderRepository = $orderRepository;
    }

    /**
     * @Given A service has been successfully injected through the Context␣
↪constructor
     */
```

(continues on next page)

(continued from previous page)

```
    public function␣
↪aServiceHasBeenSuccessfullyInjectedThroughTheContextConstructor()
    {
        if (!$this->orderRepository instanceof OrderRepositoryInterface) {
            throw new Exception('Something went wrong :(');
        }
    }

    /**
     * @When I work with Behat
     */
    public function iWorkWithBehat()
    {
        // no-op
    }

    /**
     * @Then I am happy
     */
    public function iAmHappy()
    {
        // no-op :)
    }
}
```

Note that here we inject the Order Repository Magento service through the Context constructor, but it is also possible to inject it through the Behat Step definition as well. For more information see the *usage section of this documentation*.

Run Behat and you should see the test passing.

```
$ bin/behat features/my_feature.feature
```

1.2 Installation

1.2.1 Requirements

- PHP 7.1+
- Behat 3.5+
- Magento 2 2.2+

1.2.2 Using Composer

The recommended installation method is through Composer:

```
$ composer require --dev bex/behat-magento2-extension
```

1.3 Configuration

1.3.1 Enable the Extension

You can enable the extension in your `behat.yml` in following way:

```
default:
  extensions:
    Bex\Behat\Magento2Extension: ~
```

1.3.2 Configure the Service Container

In order to be able to access the Magento 2 services from your Behat Contexts you need to configure the Magento2 Behat Service Container for your test suite. You can do it like this:

```
default:
  suites:
    yoursuite:
      services: '@bex.magento2_extension.service_container'
```

With the above configuration Behat will use the service container provided by this extension which makes all services defined in the Magento 2 DI available to inject into any Context.

Note that you need to pass over the dependencies to your Contexts manually like this:

```
default:
  suites:
    yoursuite:
      contexts:
        - YourContext:
          - '@Magento\Catalog\Api\ProductRepositoryInterface'

      services: '@bex.magento2_extension.service_container'
```

1.3.3 Enable Autowiring for Contexts

This extension does not override the default Behat argument resolvers. Because of this you can take advantage of the default Behat service autowiring feature. You can enable this feature by adding `autowire: true` to the behat config of your test suite. After that services from Magento will be automatically injected to the Contexts without any manual configuration.

```
default:
  suites:
    yoursuite:
      autowire: true

      contexts:
        - YourContext

      services: '@bex.magento2_extension.service_container'
```

Note that the argument resolver is able to autowire services for:

- constructor arguments

- step definition arguments
- transformation arguments

For more information see the *usage examples section of this documentation*.

1.3.4 Configure the Magento area

Services in the Magento DI can be defined on global level (in any module's `etc/di.xml`) but you can also define and/or override services for a specific Magento area (e.g. `etc/frontend/di.xml`). When testing your feature you might want to access services defined for a specific area so in order to support this the extension provides an additional config option which you can change per test suite. You can configure the required area in the following way:

```
default:
  suites:
    yoursuite:
      contexts:
        - YourContext

      services: '@bex.magento2_extension.service_container'

      magento:
        area: adminhtml
```

This will tell the extension to load the services from the `adminhtml` area. Note that by default only the `global` area services are loaded. When specifying an area in the config you will always get all services from the `global` area extended by the specific configured area. For example in the above case you will get all the services from the `global` area overridden/extended by the `adminhtml` area.

If you have a `test` area (see Configure Magento DI overrides section) then you can configure it here as area. Also if you would like to use a built-in area in combination with the test area then you can configure it like this:

```
magento:
  area: [adminhtml, test]
```

The extension will take care of the loading and merging of the service configurations of these areas in the provided order. So in the above example the following will happen:

1. `global` area is loaded
2. `adminhtml` area is loaded and overrides services / adds new services
3. `test` area is loaded and overrides services / adds new services

1.3.5 Configure Magento DI overrides

When you test your feature you might want to mock some services to e.g. avoid using external services like database, cache, etc. during your test run.

In order to achieve this we can use a custom Magento area where we can easily replace dependencies.

To do this we need to do 3 things:

1. Configure a new test area in Magento

This can be done by defining the custom area in your module's `etc/di.xml` in the following way:

```
<?xml version="1.0" encoding="utf-8"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"␣
↪xsi:noNamespaceSchemaLocation="urn:magento:framework:ObjectManager/etc/config.xsd">
    <type name="Magento\Framework\App\AreaList">
        <arguments>
            <argument name="areas" xsi:type="array">
                <item name="test" xsi:type="null" />
            </argument>
        </arguments>
    </type>
</config>
```

Alternatively you can install the Test area Magento 2 module which will define an area called `test` for you, so you can do the di overrides in your module's `etc/test/di.xml`.

Note: Don't forget to clear the Magento cache to reload the available area codes.

2. Define custom DI configuration in that area

Since the `test` area now exsits as a valid area code in Magento, you can freely change any DI configuration in your module's *etc/test/di.xml*. E.g.:

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"␣
↪xsi:noNamespaceSchemaLocation="urn:magento:framework:ObjectManager/etc/config.xsd">
    <preference for="Magento\Catalog\Api\ProductRepositoryInterface" type=
↪"Foo\Bar\Test\FakeProductRepository" />
</config>
```

3. Configure this test area in the Behat suite configuration

In order to load this custom DI configuration during the test run the test area need to be configured in the Behat test suite so it can load to merge it with the default area.

```
default:
  suites:
    yoursuite:
      autowire: true

      contexts:
        - YourContext

      services: '@bex.magento2_extension.service_container'

      magento:
        area: test
```

Note that the above configuration will only load services from the `global` and `test` areas. If you would like to load services from another area as well (e.g. `adminhtml`) then you can specify the a list of area codes as parameter for the `area` config option. For more information see the "Configure the Magento area" section above.

And that's all. If you inject a service into your Context which uses the `ProductRepositoryInterface` or inject the `ProductRepositoryInterface` itself then the `FakeProductRepository` will be used as its dependency instead of the default `ProductRepository`.

1.3.6 Configure Behat Helper Services

If you are familiar with the helper container feature in Behat then probably you already got used to defining helper services under the `services` configuration key like this:

```
default:
  suites:
    default:
      contexts:
        - FirstContext:
          - "@SharedService"
        - SecondContext:
          - "@SharedService"

      services:
        SharedService: ~
```

Unfortunately the custom service container is registered under the same key (see Configure the Service Container section) so we are not able to specify our helper services here. But don't worry this extension allows you to register your helper services in a custom Symfony DI container in the following way:

1. Configure the path for the service container configuration file:

```
default:
  extensions:
    Bex\Behat\Magento2Extension:
      services: features/bootstrap/config/services.yml
```

Note: You can use `yml`, `xml` or `php` format. For more information see the official documentation of the Symfony DI component.

2. Define your helper service:

Define your helper services in the servies configuration file which you created in the first step.

```
services:
  _defaults:
    public: true

  SharedService: ~
```

3. Inject your helper service into your Behat Context:

```
default:
  suites:
    yoursuite:
      contexts:
        - YourContext:
          - '@Magento\Catalog\Api\ProductRepositoryInterface'
          - '@SharedService'
      services: '@bex.magento2_extension.service_container'
```

Alternatively if you are using autowiring (see Enable Autowiring for Contexts section) then you can skip this step since the context arguments will be autowired even from this custom Symfony service container.

That's all. Now your helper service should be successfully injected to your Behat Context.

1.3.7 Inject dependencies to helper services

Since the helper services are defined in a custom Symfony DI container (see Configure Behat Helper Services section) it is possible to pass over dependencies to your helper services. You can simply do this in the following way:

```
services:
  _defaults:
    public: true

  AnotherSharedService: ~

  SharedService:
    arguments: ['@AnotherSharedService']
```

In addition to this the extension gives you access to any service defined in the default Behat service container or in the Magento DI. Which means you can inject any service defined by the Behat application itself or by any Behat extension or by Magento into your helper services.

```
services:
  _defaults:
    public: true

  AnotherSharedService: ~

  SharedService:
    arguments:
      - '@AnotherSharedService'
      - '@Magento\Sales\Api\OrderRepositoryInterface'
      - '@mink'
      - '%paths.base%'
```

In the above example we injected services from various places:

- `@AnotherSharedService` injected from the helper service container
- `@Magento\Sales\Api\OrderRepositoryInterface` injected from the Magento DI
- `@mink` injected from the MinkExtension (this example only works if you have the MinkExtension extension installed)
- `%paths.base%` injected from the Behat built-in service container

1.3.8 Enable Autowiring for helper services

The helper services are defined in the custom Symfony DI container (see Configure Behat Helper Services section) so we can take advantage of the autowire feature of the Symfony DI component as well. You can enable this feature by adding the `autowire: true` configuration to your service container configuration.

```
services:
  _defaults:
    public: true
    autowire: true

  AnotherSharedService: ~

  SharedService:
    arguments:
```

(continues on next page)

(continued from previous page)

```
      $mink: '@mink'
      $basePath: '%paths.base%'
```

As you can see all injectable service argument omitted. But we still need to specify 2 arguments: - Mink service cannot be autowired because the service id is not the FQCN - Base Path cannot be autowired since it is a string parameter

1.3.9 Configure the Magento bootstrap path

If your Magento `bootstrap.php` is not available in the default `app/bootstrap.php` location then you can specify the custom path in the following way:

```
default:
  extensions:
    Bex\Behat\Magento2Extension:
      bootstrap: path/to/your/bootstrap.php # by default app/bootstrap.php
```

1.4 Usage Examples

1.4.1 Manually inject service to Behat Context as constructor argument

If you didn't enable the Behat autowire feature then you need to provide your Behat Context dependencies manually in the Behat config. E.g.:

Feature:

```
Feature: Magento and Behat DI connected
  As a developer
  In order to write Behat tests easily
  I should be able to inject services from the Magento DI into Behat Contexts

  Scenario: Injecting service from Magento DI to Behat Context as argument for Behat␣
↪Context constructor
    Given A service has been successfully injected through the Context constructor
    When I work with Behat
    Then I am happy
```

Context:

```
<?php

use Behat\Behat\Context\Context;
use Exception;
use Magento\Catalog\Api\ProductRepositoryInterface;

class FeatureContext implements Context
{
    /** @var ProductRepositoryInterface */
    private $productRepository;

    public function __construct(ProductRepositoryInterface $productRepository)
    {
        $this->productRepository = $productRepository;
```

(continues on next page)

(continued from previous page)

```
    }

    /**
     * @Given A service has been successfully injected through the Context constructor
     */
    public function theProductRepositorySuccessfullyInjectedAsConstructorArgument()
    {
        if (!$this->productRepository instanceof ProductRepositoryInterface) {
            throw new Exception('Something went wrong :(');
        }
    }

    /**
     * @When I work with Behat
     */
    public function iWorkWithBehat()
    {
        // no-op
    }

    /**
     * @Then I am happy
     */
    public function iAmHappy()
    {
        // no-op :)
    }
}
```

Configuration:

```
default:
  suites:
    yoursuite:
      contexts:
        - YourContext:
          - '@Magento\Catalog\Api\ProductRepositoryInterface'
      services: '@bex.magento2_extension.service_container'
```

That's all. With the above the Product Repository will be injected to your Behat Context.

1.4.2 Automatically inject service to Behat Context as constructor argument

You can enable the Behat service autowiring feature to get the services automatically injected to Contexts. E.g.:

Feature:

```
Feature: Magento and Behat DI connected
  As a developer
  In order to write Behat tests easily
  I should be able to inject services from the Magento DI into Behat Contexts

  Scenario: Injecting service from Magento DI to Behat Context as argument for Behat␣
↪Context constructor
    Given A service has been successfully injected through the Context constructor
```

(continues on next page)

(continued from previous page)

```
    When I work with Behat
    Then I am happy
```

Context:

```
<?php

use Behat\Behat\Context\Context;
use Exception;
use Magento\Catalog\Api\ProductRepositoryInterface;

class FeatureContext implements Context
{
    /** @var ProductRepositoryInterface */
    private $productRepository;

    public function __construct(ProductRepositoryInterface $productRepository)
    {
        $this->productRepository = $productRepository;
    }

    /**
     * @Given A service has been successfully injected through the Context constructor
     */
    public function theProductRepositorySuccessfullyInjectedAsConstructorArgument()
    {
        if (!$this->productRepository instanceof ProductRepositoryInterface) {
            throw new Exception('Something went wrong :(');
        }
    }

    /**
     * @When I work with Behat
     */
    public function iWorkWithBehat()
    {
        // no-op
    }

    /**
     * @Then I am happy
     */
    public function iAmHappy()
    {
        // no-op :)
    }
}
```

Configuration:

```
default:
  suites:
    yoursuite:
      autowire: true

      contexts:
        - YourContext
```

(continues on next page)

(continued from previous page)

```
      services: '@bex.magento2_extension.service_container'
```

1.4.3 Inject service to Behat Context as Behat Step argument

The Behat service autowiring feature allows to inject services from the configured service container to any of the Step Definitions as argument. You can use this feature in combination with this extension as well. E.g.:

Feature:

```
Feature: Magento and Behat DI connected
  As a developer
  In order to write Behat tests easily
  I should be able to inject services from the Magento DI into Behat Contexts

  Scenario: Injecting service from Magento DI to Behat Context as argument for Behat␣
↪Step
    Given A service has been successfully injected as argument to this step
    When I work with Behat
    Then I am happy
```

Context:

```
<?php

use Behat\Behat\Context\Context;
use Magento\Catalog\Api\ProductRepositoryInterface;

class YourContext implements Context
{
    /**
     * @Given A service has been successfully injected as argument to this step
     */
    public function␣
↪theProductRepositorySuccessfullyInjectedAsArgument(ProductRepositoryInterface
↪$productRepository)
    {
        if (!$this->productRepository instanceof ProductRepositoryInterface) {
            throw new Exception('Something went wrong :(');
        }
    }
}
```

Configuration:

```
default:
  suites:
    yoursuite:
      autowire: true

      contexts:
        - YourContext

      services: '@bex.magento2_extension.service_container'
```

1.4.4 Inject service to Behat Context as Behat Step Argument Transformer argument

The Behat service autowiring feature allows to inject services from the configured service container to any of the Step Argument Transformer method as argument. You can use this feature in combination with this extension as well. E.g.:

Feature:

```
Feature: Magento and Behat DI connected
  As a developer
  In order to write Behat tests easily
  I should be able to inject services from the Magento DI into Behat Contexts

  Scenario: Injecting service from Magento DI to Behat Context as argument for Behat␣
↪Step Paramater Transformation method
    Given A service has been successfully injected to the parameter transformation␣
↪method while transforming "foobar"
    When I work with Behat
    Then I am happy
```

Context:

```
<?php

use Behat\Behat\Context\Context;
use Magento\Catalog\Api\Data\ProductInterface;
use Magento\Catalog\Api\Data\ProductInterfaceFactory as ProductFactory;
use Magento\Catalog\Api\ProductRepositoryInterface;

class YourContext implements Context
{
    /**
     * @Transform
     */
    public function transformStringToProduct(
        string $productSku,
        ProductRepositoryInterface $productRepository,
        ProductFactory $productFactory
    ): ProductInterface {
        Assert::assertInstanceOf(ProductRepositoryInterface::class,
↪$productRepository);

        try {
            return $productRepository->get($productSku);
        } catch (NoSuchEntityException $e) {
            // product does not exsits - normally you would let the test fail here
            // but for this demonstration we will just create a new product in memory
            // also note that the product factory autogenerated even when it is␣
↪requested from Behat
            return $productFactory->create()->setSku($productSku);
        }
    }

    /**
     * @Given A service has been successfully injected to the parameter␣
↪transformation method while transforming :product
     */
    public function theProductSkuSuccessFullyTransformedToProduct(ProductInterface
↪$product)
```

(continues on next page)

(continued from previous page)

```
    {
        if (!$product instanceof ProductInterface) {
            throw new Exception('Something went wrong :(');
        }
    }
}
```

Configuration:

```
default:
  suites:
    yoursuite:
      autowire: true

      contexts:
        - YourContext

      services: '@bex.magento2_extension.service_container'
```

1.4.5 Mocking Dependency

Given you have an application service interface like this:

```
<?php

namespace Vendor\ModuleName\Config;

interface ConfigProviderInterface
{
    public function isFreeDeliverEnabled(): bool;

    public function getFreeDeliveryThreshold(): float;
}
```

And you have an implementation for this service:

```
<?php

namespace Vendor\ModuleName\Config;

use Magento\Framework\App\Config\ScopeConfigInterface;

class ConfigProvider implements ConfigProviderInterface
{
    /** @var ScopeConfigInterface */
    private $scopeConfig;

    public function __construct(ScopeConfigInterface $scopeConfig)
    {
        $this->scopeConfig = $scopeConfig;
    }

    public function isFreeDeliverEnabled(): bool
    {
        return $this->scopeConfig->isSetFlag('path/to/config');
```

(continues on next page)

(continued from previous page)

```
    }

    public function getFreeDeliveryThreshold(): float
    {
        return (float) $this->scopeConfig->getValue('path/to/another/config');
    }
}
```

And you have the following DI config to mark this implementation as the default implementation:

```
<preference for="Vendor\ModuleName\Config\ConfigProviderInterface" type=
↪"Vendor\ModuleName\Config\ConfigProvider" />
```

In addition to these you have an application service which depends on this config interface, e.g.:

```
<?php

namespace Vendor\ModuleName\Service;

use Magento\Quote\Model\Quote;

class DeliveryCostCalculator implements DeliveryCostCalculatorInterface
{
    private const DELIVERY_COST = 5.0;

    /** @var ConfigProviderInterface */
    private $deliveryConfig;

    public function __construct(ConfigProviderInterface $deliveryConfig)
    {
        $this->deliveryConfig = $deliveryConfig;
    }

    public function calculate(Quote $quote): float
    {
        if (!$this->deliveryConfig->isFreeDeliverEnabled()) {
            return self::DELIVERY_COST;
        }

        if ($quote->getGrandTotal() < $this->deliveryConfig->
↪getFreeDeliveryThreshold()) {
            return self::DELIVERY_COST;
        }

        return 0.0;
    }
}
```

When you write your application tests, if you would like to avoid relying on the database, then you either need to mock `Magento\Framework\App\Config\ScopeConfigInterface` or `Vendor\ModuleName\Config\ConfigProviderInterface`. Lets assume we would like to mock our own `ConfigProviderInterface` this time.

First of all we need to configure a `test` area in Magento. We can do this by adding the following to the module's global `etc/di.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"␣
↪xsi:noNamespaceSchemaLocation="urn:magento:framework:ObjectManager/etc/config.xsd">
    <type name="Magento\Framework\App\AreaList">
        <arguments>
            <argument name="areas" xsi:type="array">
                <item name="test" xsi:type="null" />
            </argument>
        </arguments>
    </type>
</config>
```

Or we can simply install the Test area Magento 2 module which will define an area called `test` in the same way. :)

Now we can define our DI overrides in the module's `etc/test/di.xml`.

It will look like this:

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"␣
↪xsi:noNamespaceSchemaLocation="urn:magento:framework:ObjectManager/etc/config.xsd">
    <preference for="Vendor\ModuleName\Config\ConfigProviderInterface" type=
↪"Vendor\ModuleName\Test\FakeConfigProvider" />
</config>
```

And we are done. After a cache clear everything should be ready to use. If you inject the `Vendor\ModuleName\Service\DeliveryCostCalculator` into your Behat Context then it will use the `Vendor\ModuleName\Test\FakeConfigProvider` which we can freely modify in our tests.

E.g.:

FakeConfigProvider:

```
<?php

namespace Vendor\ModuleName\Test;

use Magento\Framework\App\Config\ScopeConfigInterface;

class FakeConfigProvider implements ConfigProviderInterface
{
    /** @var bool */
    private $isFreeDeliveryEnabled = false;

    /** @var float */
    private $freeDeliveryThreshold = 0.0;

    public function isFreeDeliverEnabled(): bool
    {
        return $this->isFreeDeliveryEnabled;
    }

    public function getFreeDeliveryThreshold(): float
    {
        return (float) $this->freeDeliveryThreshold;
    }

    public function enableFreeDelivery(): void
    {
```

(continues on next page)

(continued from previous page)

```
        $this->isFreeDeliveryEnabled = true;
    }

    public function disableFreeDelivery(): void
    {
        $this->isFreeDeliveryEnabled = false;
    }

    public function setFreeDeliveryThreshold(float $threshold): void
    {
        $this->freeDeliveryThreshold = $threshold;
    }
}
```

DeliveryContext:

```
<?php

use Behat\Behat\Context\Context;
use Behat\Gherkin\Node\TableNode;
use Exception;
use Vendor\ModuleName\Service\DeliveryCostCalculator;
use Vendor\ModuleName\Test\FakeConfigProvider;

class DeliveryContext implements Context
{
    /** @var DeliveryCostCalculator */
    private $deliveryCostCalculator;

    /** @type float|null */
    private $deliveryCost = null;

    public function __construct(DeliveryCostCalculator $deliveryCostCalculator)
    {
        $this->deliveryCostCalculator = $deliveryCostCalculator;
    }

    /**
     * @Given The cart contains the following items:
     */
    public function theCartContainsTheFollowingItems(TableNode $table)
    {
        // Create a Cart here
        // $this->currentQuote = ...
    }

    /**
     * @Given The free delivery is enabled
     */
    public function theFreeDeliveryIsEnabled(FakeConfigProvider $deliveryConfig)
    {
        $deliveryConfig->enableFreeDelivery();
    }

    /**
     * @Given The free delivery is disabled
     */
```

(continues on next page)

(continued from previous page)

```
    public function theFreeDeliveryIsDisabled(FakeConfigProvider $deliveryConfig)
    {
        $deliveryConfig->disableFreeDelivery();
    }

    /**
     * @Given The free delivery cost threshold is configured to :threshold
     */
    public function theFreeDeliveryCostThresholdIsConfiguredTo(float $threshold,␣
↪FakeConfigProvider $deliveryConfig)
    {
        $deliveryConfig->setFreeDeliveryThreshold($threshold);
    }

    /**
     * @When The delivery cost is calculated
     */
    public function theDeliveryCostIsCalculated()
    {
        $this->deliveryCost = $this->deliveryCostCalculator->calculate($this->
↪currentQuote);
    }

    /**
     * @Then The delivery cost is :expectedDeliveryCost
     */
    public function theDeliveryCostIs(float $expectedDeliveryCost)
    {
        if ($expectedDeliveryCost !== $this->deliveryCost) {
            throw new Exception(
                spritf('Delivery cost expected to be %s but got %s',
↪$expectedDeliveryCost, $this->deliveryCost)
            );
        }
    }
}
```

The above context is not complete, it is just an example to show how easy to mock the dependencies this way.

CHAPTER 2

References

- Github Repository: https://github.com/tkotosz/BehatMagento2Extension
- Packagist: https://packagist.org/packages/bex/behat-magento2-extension
- Behat Official Documentaion: https://docs.behat.org