Payment Suite

Release

Apr 25, 2017

Contents

The `Payment Suite Project` is just a way to implement any payment platform using some Symfony2 components, with a common structure. Your project will simply need to listen to a few events, so the payment platform usage will be fully transparent.

CHAPTER 1

User Documentation

Installation

You have to add require line into you composer.json file. Notice that you need to override the defined platform name with desired one.

You have to add require line into you composer.json file

Note: you need to replace `platform` with bundle name you want to install

```
"require": {
    // ...
    "paymentsuite/platform-bundle": "X.X.X"
}
```

Then you have to use composer to update your project dependencies

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar update
```

And register the bundle in your `AppKernel.php` file

```
return array(
    // ...
    new PaymentSuite\PaymentCoreBundle\PaymentCoreBundle(),
    new PaymentSuite\PlatformBundle\PlatformBundle(),
);
```

Configuration

About PaymentBridgeBundle

As Payment Suite should be compatible with all E-commerces projects, it's built without any kind of attachment with your model, so you must build (just once) a specific bridge bundle to tell Payment Suite where to find some data.

For this purpose, create a Bundle named PaymentBridgeBundle with the following features.

PaymentBridge Service

Payment Service is a service that has to be necessarily implemented. This service **must** implement `PaymentSuite\PaymentCoreBundle\Services\Interfaces\PaymentBridgeInterface`.

```php
<?php

namespace YourProjectName\PaymentCoreBundle\Services;

use PaymentSuite\PaymentCoreBundle\Services\Interfaces\PaymentBridgeInterface;

class PaymentBridge implements PaymentBridgeInterface
{
    /**
     * @var Object
     *
     * Order object
     */
    private $order;

    /**
     * Set order to PaymentBridge
     *
     * @var Object $order Order element
     */
    public function setOrder($order)
    {
        $this->order = $order;
    }

    /**
     * Return order
     *
     * @return Object order
     */
    public function getOrder()
    {
        return $this->order;
    }

    /**
     * Return order identifier value
     *
     * @return integer
     */
    public function getOrderId()
    {
```

```
        return $this->order->getId();
    }

    /**
     * Given an id, find Order
     *
     * @return Object order
     */
    public function findOrder($orderId)
    {
        /*
        * Your code to get Order
        */

        return $this->order;
    }

    /**
     * Get the currency in which the order is paid
     *
     * @return string
     */
    public function getCurrency()
    {
        /*
        * Set your static or dynamic currency
        */

        return 'USD';
    }

    /**
     * Get total order amount in cents
     *
     * @return integer
     */
    public function getAmount()
    {
        /*
        * Return payment amount (in cents)
        */

        return $amount;
    }

    /**
     * Return if order has already been paid
     *
     * @return boolean
     */
    public function isOrderPaid()
    {
        return array();
    }

    /**
     * Get extra data
     *
```

```
     * @return array
     */
    public function getExtraData()
    {
        return false;
    }
}
```

This service **must** be named `payment.bridge` and configured in the `Resources\config\services.yml` file:

```
services:
    # ...
    payment.bridge:
        class: YourProjectName\PaymentBridgeBundle\Services\PaymentBridge
```

Payment Event Listener

You can create an Event Listener to subscribe to Payment process events.

In fact, this will be the way to manage your cart and your order in every payment stage.

```
<?php

namespace YourProjectName\PaymentBridgeBundle\EventListener;

use PaymentSuite\PaymentCoreBundle\Event\PaymentOrderLoadEvent;
use PaymentSuite\PaymentCoreBundle\Event\PaymentOrderCreatedEvent;
use PaymentSuite\PaymentCoreBundle\Event\PaymentOrderDoneEvent;
use PaymentSuite\PaymentCoreBundle\Event\PaymentOrderSuccessEvent;
use PaymentSuite\PaymentCoreBundle\Event\PaymentOrderFailEvent;

/**
 * Payment event listener
 *
 * This listener is enabled whatever the payment method is.
 */
class Payment
{
    /**
     * On payment order load event
     *
     * @param PaymentOrderLoadEvent $paymentOrderLoadEvent Payment Order Load event
     */
    public function onPaymentOrderLoad(PaymentOrderLoadEvent $paymentOrderLoadEvent)
    {
        /*
         * Your code for this event
         */
    }

    /**
     * On payment order created event
     *
     * @param PaymentOrderCreatedEvent $paymentOrderCreatedEvent Payment Order␣
↪Created event
     */
```

```
    public function onPaymentOrderCreated(PaymentOrderCreatedEvent
↪$paymentOrderCreatedEvent)
    {
        /*
         * Your code for this event
         */
    }

    /**
     * On payment done event
     *
     * @param PaymentOrderDoneEvent $paymentOrderDoneEvent Payment Order Done event
     */
    public function onPaymentDone(PaymentOrderDoneEvent $paymentOrderDoneEvent)
    {
        /*
         * Your code for this event
         */
    }

    /**
     * On payment success event
     *
     * @param PaymentOrderSuccessEvent $paymentOrderSuccessEvent Payment Order␣
↪Success event
     */
    public function onPaymentSuccess(PaymentOrderSuccessEvent
↪$paymentOrderSuccessEvent)
    {
        /*
         * Your code for this event
         */
    }

    /**
     * On payment fail event
     *
     * @param PaymentOrderFailEvent $paymentOrderFailEvent Payment Order Fail event
     */
    public function onPaymentFail(PaymentOrderFailEvent $paymentOrderFailEvent)
    {
        /*
         * Your code for this event
         */
    }
}
```

Register these event listeners in your `Resources\config\services.yml` file:

```
services:
    # ...
    payment.event.listener:
        class:     YourProjectName\PaymentBridgeBundle\EventListener\Payment
        arguments: [@doctrine.orm.entity_manager, @mailer]
        tags:
            - { name: kernel.event_listener, event: payment.order.done, method:␣
↪onPaymentOrderDone }
            - { name: kernel.event_listener, event: payment.order.created, method:␣
↪onPaymentOrderCreated }
```

```
            - { name: kernel.event_listener, event: payment.order.load, method:␣
↪onPaymentLoad }
            - { name: kernel.event_listener, event: payment.order.success, method:␣
↪onPaymentSuccess }
            - { name: kernel.event_listener, event: payment.order.fail, method:␣
↪onPaymentFail }
```

Available Platforms

These are the platforms currently supported by the main PaymentSuite team. Each platform has been developed as a wrapper of PaymentCoreBundle

AuthorizenetBundle

This bundle bring you a possibility to make simple payments through Authorize.Net.

Install

You have to add require line into you composer.json file

```
"require": {
    // ...
    "paymentsuite/authorizenet-bundle": "v1.1"
}
```

Then you have to use composer to update your project dependencies

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar update paymentsuite/authorizenet-bundle
```

And register the bundle in your `AppKernel.php` file

```
return array(
    // ...
    new PaymentSuite\PaymentCoreBundle\PaymentCoreBundle(),
    new PaymentSuite\AuthorizenetBundle\AuthorizenetBundle(),
);
```

Configuration

If it's first payment method of PaymentSuite in your project, first you have to configure PaymentBridge Service and Payment Event Listener following this documentation.

Configure the AuthorizenetBundle parameters in your `config.yml`.

```
authorizenet:

    # authorizenet keys
    login_id: XXXXXXXXXXX
    tran_key: XXXXXXXXXXX
    test_mode: true
```

```
   # By default, controller route is /payment/authorizenet/execute
   controller_route: /my/custom/route

   # Configuration for payment success redirection
   #
   # Route defines which route will redirect if payment success
   # If order_append is true, Bundle will append cart identifier into route
   #    taking order_append_field value as parameter name and
   #    PaymentOrderWrapper->getOrderId() value
   payment_success:
      route: cart_thanks
      order_append: true
      order_append_field: order_id

   # Configuration for payment fail redirection
   #
   # Route defines which route will redirect if payment fails
   # If cart_append is true, Bundle will append cart identifier into route
   #    taking cart_append_field value as parameter name and
   #    PaymentCartWrapper->getCartId() value
   payment_fail:
      route: cart_view
      cart_append: false
      cart_append_field: cart_id
```

About Authorizenet `login_id` and `tran_key` you can learn more in Authorizenet documentation page.

Router

AuthorizenetBundle allows developer to specify the route of controller where Authorize.Net callback is processed. By default, this value is `/payment/authorizenet/callback` but this value can be changed in configuration file. Anyway AuthorizenetBundle's routes must be parsed by the framework, so these lines must be included into `routing.yml` file.

```
authorizenet_payment_routes:
   resource: .
   type: authorizenet
```

Display

Once your AuthorizenetBundle is installed and well configured, you need to place your payment form.

AuthorizenetBundle gives you all form view as requested by the payment module.

```
{% block content %}
   <div class="payment-wrapper">
      {{ authorizenet_render() }}
   </div>
{% endblock content %}
```

Customize

`authorizenet_render()` just print a basic form.

As every project need its own form design, you can overwrite default form located in: `app/Resources/AuthorizenetBundle/views/Authorizenet/view.html.twig`.

Testing and more documentation

For testing you can use these example these examples. More detail about Authorizenet API you can find in this web.

GoogleWalletBundle

This bundle bring you a possibility to make simple payments through Google Wallet.

Install

You have to add require line into you composer.json file

```
"require": {
    // ...
    "paymentsuite/google-wallet-bundle": "v1.1"
}
```

Then you have to use composer to update your project dependencies

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar update paymentsuite/google-wallet-bundle
```

And register the bundle in your `AppKernel.php` file

```
return array(
    // ...
    new PaymentSuite\PaymentCoreBundle\PaymentCoreBundle(),
    new PaymentSuite\GoogleWalletBundle\GoogleWalletBundle(),
);
```

Configuration

If it's first payment method of PaymentSuite in your project, first you have to configure PaymentBridge Service and Payment Event Listener following this documentation.

Configure the GoogleWalletBundle parameters in your `config.yml`.

```
google_wallet:

    # google wallet keys
    merchant_id: XXXXXXXXXXXX
    secret_key: XXXXXXXXXXXX

    # Configuration for payment success redirection
    #
    # Route defines which route will redirect if payment success
    # If order_append is true, Bundle will append cart identifier into route
    #    taking order_append_field value as parameter name and
    #    PaymentOrderWrapper->getOrderId() value
    payment_success:
        route: cart_thanks
```

```
        order_append: true
        order_append_field: order_id

    # Configuration for payment fail redirection
    #
    # Route defines which route will redirect if payment fails
    # If cart_append is true, Bundle will append cart identifier into route
    #    taking cart_append_field value as parameter name and
    #    PaymentCartWrapper->getCartId() value
    payment_fail:
        route: cart_view
        cart_append: false
        cart_append_field: cart_id
```

To get `merchant_id` and `secret_key` you have to register for Sandbox Settings or Production Settings. Also there you have to set postback URL (must be on public DNS and not localhost). For more information you can visit page of Google Wallet APIs.

Extra Data

PaymentBridge Service must return, at least, these fields.

- order_name
- order_description

Router

GoogleWalletBundle allows developer to specify the route of controller where Google Wallet callback is processed. By default, this value is `/payment/googlewallet/callback` but this value can be changed in configuration file. Anyway GoogleWalletBundle's routes must be parsed by the framework, so these lines must be included into `routing.yml` file.

```
google_wallet_payment_routes:
    resource: .
    type: googlewallet
```

Display

Once your GoogleWalletBundle is installed and well configured, you need to place submit button which open Google Wallet pop-up.

GoogleWalletBundle gives you all code as requested by the payment module.

```
{% block content %}
    <div class="payment-wrapper">
        {{ googlewallet_render() }}
    </div>
{% endblock content %}

{% block foot_script %}
    {{ parent() }}
    {{ googlewallet_scripts() }}
{% endblock foot_script %}
```

Customize

As every project need its own form design, you can overwrite default button located in: `app/Resources/GoogleWalletBundle/views/GoogleWallet/view.html.twig`.

Testing and more documentation

For testing, you just have to use sandbox settings. More details about Google Wallet API you can find in this web.

PaymillBundle

Configuration

Configure the PaymillBundle configuration in your `config.yml`

```
paymill:

    # paymill keys
    public_key: XXXXXXXXXXXX
    private_key: XXXXXXXXXXXX

    # By default, controller route is /payment/paymill/execute
    controller_route: /my/custom/route

    # Configuration for payment success redirection
    #
    # Route defines which route will redirect if payment successes
    # If order_append is true, Bundle will append card identifier into route
    #    taking order_append_field value as parameter name and
    #    PaymentOrderWrapper->getOrderId() value
    payment_success:
        route: card_thanks
        order_append: true
        order_append_field: order_id

    # Configuration for payment fail redirection
    #
    # Route defines which route will redirect if payment fails
    # If card_append is true, Bundle will append card identifier into route
    #    taking card_append_field value as parameter name and
    #    PaymentCardWrapper->getCardId() value
    payment_fail:
        route: card_view
        card_append: false
        card_append_field: card_id
```

Extra Data

PaymentBridge Service must return, at least, these fields.

- order_description

Router

PaymillBundle allows developer to specify the route of controller where paymill payment is processed. By default, this value is `/payment/paymill/execute` but this value can be changed in configuration file. Anyway, the bundle routes must be parsed by the framework, so these lines must be included into routing.yml file

Display

Once your Paymill is installed and well configured, you need to place your payment form.

PaymillBundle gives you all form view as requested by the payment module.

Customize

`paymill_render()` only print form in a simple way.

As every project need its own form design, you should overwrite in `app/Resources/PaymillBundle/views/Paymill/view.html.twig`, paymill form render template placed in `PaymentSuite/Paymill/Bundle/Resources/views/Paymill/view.html.twig`.

StripeBundle

This bundle bring you a possibility to make simple payments through Stripe.

Install

You have to add require line into you composer.json file

```
"require": {
    // ...
    "paymentsuite/stripe-bundle": "v1.1"
}
```

Then you have to use composer to update your project dependencies

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar update paymentsuite/stripe-bundle
```

And register the bundle in your `AppKernel.php` file

```
return array(
    // ...
    new PaymentSuite\PaymentCoreBundle\PaymentCoreBundle(),
    new PaymentSuite\StripeBundle\StripeBundle(),
);
```

Configuration

If it's first payment method of PaymentSuite in your project, first you have to configure PaymentBridge Service and Payment Event Listener following this documentation.

Configure the StripeBundle parameters in your `config.yml`.

```
stripe:

    # stripe keys
    public_key: XXXXXXXXXXXX
    private_key: XXXXXXXXXXXX

    # By default, controller route is /payment/stripe/execute
    controller_route: /my/custom/route

    # Configuration for payment success redirection
    #
    # Route defines which route will redirect if payment success
    # If order_append is true, Bundle will append cart identifier into route
    #    taking order_append_field value as parameter name and
    #    PaymentOrderWrapper->getOrderId() value
    payment_success:
        route: cart_thanks
        order_append: true
        order_append_field: order_id

    # Configuration for payment fail redirection
    #
    # Route defines which route will redirect if payment fails
    # If cart_append is true, Bundle will append cart identifier into route
    #    taking cart_append_field value as parameter name and
    #    PaymentCartWrapper->getCartId() value
    payment_fail:
        route: cart_view
        cart_append: false
        cart_append_field: cart_id
```

About Stripe `public_key` and `private_key` you can learn more in Stripe documentation page.

Router

StripeBundle allows developer to specify the route of controller where Stripe callback is processed. By default, this value is `/payment/stripe/callback` but this value can be changed in configuration file. Anyway StripeBundle's routes must be parsed by the framework, so these lines must be included into `routing.yml` file.

```
stripe_payment_routes:
    resource: .
    type: stripe
```

Display

Once your StripeBundle is installed and well configured, you need to place your payment form.

StripeBundle gives you all form view as requested by the payment module.

```
{% block content %}
    <div class="payment-wrapper">
        {{ stripe_render() }}
    </div>
{% endblock content %}
```

```
{% block foot_script %}
    {{ parent() }}
    {{ stripe_scripts() }}
{% endblock foot_script %}
```

Customize

stripe_render() just print a basic form.

As every project need its own form design, you can overwrite default form located in: `app/Resources/StripeBundle/views/Stripe/view.html.twig` following Stripe documentation.

In another hand, Stripe recommend use jQuery form validator.

Testing and more documentation

For testing you can use these examples. More detail about Stripe API you can find in this web.

FAQ

CHAPTER 2

Developer Documentation

Developing Platform

Since any payment platform is implemented on the existing PaymentSuite for Symfony2 is something like a plugin, must be implemented simply those specific features of the platform itself.

The core provides a number of tools, both definition and execution, so it is not too complex to implement each of the platforms, and providing homogeneity in the set of all events regarding concerns.

PaymentMethod

The first class that must implement either integrated platform is called PaymentMethod. This must extend an interface located in `Mmoreram\PaymentCoreBundle\PaymentMethodInterface`, so you should just implement a single method.

```php
<?php

namespace Mmoreram\PaymentCoreBundle;


/**
 * Interface for all type of payments
 */
interface PaymentMethodInterface
{

    /**
     * Return type of payment name
     *
     * @return string
     */
    public function getPaymentName();
}
```

At the time that our platform offers data on the response of the payment, it is interesting that this class implements their getters, although not common on all platforms.

This is done because there may be a case where a project wants to subscribe to an event of Core, acting only if the payment is one in specific. In this case, you will have access to the data offered without any problem.

Here is an example of what could be a kind of a new payment method called AcmePaymentBundle

```php
<?php

/**
 * AcmePaymentBundle for Symfony2
 */

namespace Mmoreram\AcmePaymentBundle;

use Mmoreram\PaymentCoreBundle\PaymentMethodInterface;


/**
 * AcmePaymentMethod class
 */
class AcmePaymentMethod implements PaymentMethodInterface
{

    /**
     * @var SomeExtraData
     *
     * Some extra data given by payment response
     */
    private $someExtraData;


    /**
     * Get AcmePayment method name
     *
     * @return string Payment name
     */
    public function getPaymentName()
    {
        return 'acme_payment';
    }


    /**
     * Set some extra data
     *
     * @param string $someExtraData Some extra data
     *
     * @return AcmePaymentMethod self Object
     */
    public function setSomeExtraData($someExtraData)
    {
        $this->someExtraData = $someExtraData;

        return $this;
    }
```

```
    /**
     * Get some extra data
     *
     * @return array Some extra data
     */
    public function getSomeExtraData()
    {
        return $someExtraData;
    }
}
```

Configuration

Consider the data coming through PaymentBridge service defined by the project, and you should not redefine them statically. The configuration data is used for completely static definition. A clear example of configuration is

- Public and private keys
- API url
- Controllers routes
- Static data, like logo

This configuration must be properly defined and validated, as defined here. Let's see a configuration sample

```
services:

    acmepayment:
        public_key: XXXXXXXXXX
        private_key: XXXXXXXXXX
        payment_success:
            route: payment_success
            order_append: true
            order_append_field: order_id
        payment_fail:
            route: payment_failed
            order_append: false
```

Note: It is important to understand the motivation of configuration items. You only have to define elements unchanged at project level and environment-level writable. Pay dependent elements are placed along PaymentBridge as we will see later.

When the configuration settings are validated by the bundle, the platform should add, one by one, as parameters. Please check that all changed as a parameter fields always have the same format. Here is a short example of what could be a configuration validator.

```
<?php

/**
 * AcmePaymentBundle for Symfony2
 */

namespace Mmoreram\AcmePaymentBundle\DependencyInjection;

use Symfony\Component\Config\Definition\Builder\TreeBuilder;
```

```php
use Symfony\Component\Config\Definition\ConfigurationInterface;

/**
 * This is the class that validates and merges configuration from your app/config
↪files
 */
class Configuration implements ConfigurationInterface
{
    /**
     * {@inheritDoc}
     */
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('acmepayment');

        $rootNode
            ->children()
                ->scalarNode('public_key')
                    ->isRequired()
                    ->cannotBeEmpty()
                ->end()
                ->scalarNode('private_key')
                    ->isRequired()
                    ->cannotBeEmpty()
                ->end()
                ->arrayNode('payment_success')
                    ->children()
                        ->scalarNode('route')
                            ->isRequired()
                            ->cannotBeEmpty()
                        ->end()
                        ->booleanNode('order_append')
                            ->defaultTrue()
                        ->end()
                        ->scalarNode('order_append_field')
                            ->defaultValue('order_id')
                        ->end()
                    ->end()
                ->end()
                ->arrayNode('payment_fail')
                    ->children()
                        ->scalarNode('route')
                            ->isRequired()
                            ->cannotBeEmpty()
                        ->end()
                        ->booleanNode('order_append')
                            ->defaultTrue()
                        ->end()
                        ->scalarNode('order_append_field')
                            ->defaultValue('card_id')
                        ->end()
                    ->end()
                ->end()
            ->end();

        return $treeBuilder;
    }
```

```
}
```

And an example of parametrization of configuration items. Each platform must implement their own items.

```php
<?php

/**
 * AcmePaymentBundle for Symfony2
 */

namespace Mmoreram\AcmePaymentBundle\DependencyInjection;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\Loader;

/**
 * This is the class that loads and manages your bundle configuration
 */
class AcmePaymentExtension extends Extension
{
    /**
     * {@inheritDoc}
     */
    public function load(array $configs, ContainerBuilder $container)
    {
        $configuration = new Configuration();
        $config = $this->processConfiguration($configuration, $configs);

        $container->setParameter('acmepayment.private.key', $config['private_key']);
        $container->setParameter('acmepayment.public.key', $config['public_key']);

        $container->setParameter('acmepayment.success.route', $config['payment_success
↪']['route']);
        $container->setParameter('acmepayment.success.order.append', $config['payment_
↪success']['order_append']);
        $container->setParameter('acmepayment.success.order.field', $config['payment_
↪success']['order_append_field']);

        $container->setParameter('acmepayment.fail.route', $config['payment_fail'][
↪'route']);
        $container->setParameter('acmepayment.fail.order.append', $config['payment_
↪fail']['order_append']);
        $container->setParameter('acmepayment.fail.order.field', $config['payment_fail
↪']['order_append_field']);
    }
}
```

Extra data

All configuration of the payment must be collected by the method of `getExtraData` of `PaymentBridge` service. This method will provide all the necessary values for all installed platforms, so that each platform must, specifically, validate that the required fields are present in the method response array.

Controllers

All controller that requires payment platform itself, must be associated with a dynamically generated path. Its motivation is that the user must be able to define each of the paths associated with each of the actions of the drivers. For this, each platform must make available to the user the possibility to overwrite the path as follows.

```
<?php

namespace Mmoreram\AcmeBundle\DependencyInjection;

use Symfony\Component\Config\Definition\Builder\TreeBuilder;
use Symfony\Component\Config\Definition\ConfigurationInterface;

/**
 * This is the class that validates and merges configuration from your app/config␣
↪files
 */
class Configuration implements ConfigurationInterface
{
    /**
     * {@inheritDoc}
     */
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('acme');

        $rootNode
            ->children()
                ...

                ->scalarNode('controller_route')
                    ->defaultValue('/payment/acme/execute')
                ->end()

                ...
            ->end();

        return $treeBuilder;
    }
}
```

Once we provide the possibility to define this variable, adding one by default (should follow this pattern), we transform the variable parameter configuration, in order to inject.

```
<?php

namespace Mmoreram\AcmeBundle\DependencyInjection;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\Loader;

/**
 * This is the class that loads and manages your bundle configuration
 */
class AcmePaymentExtension extends Extension
```

```
{
    /**
     * {@inheritDoc}
     */
    public function load(array $configs, ContainerBuilder $container)
    {
        $configuration = new Configuration();
        $config = $this->processConfiguration($configuration, $configs);
        $container->setParameter('acme.controller.route', $config['controller_route
↪']);
    }
}
```

Services

All services with responsibility for launching events PaymentCore, MUST inject an instance of `Mmoreram\PaymentCoreBundle\Services\PaymentEventDispatcher`. This class is responsible for providing direct methods to launch the kernel events. All methods require paymentBridge and paymentmethod.

```
<?php

/**
 * At this point, order must be created given a card, and placed in PaymentBridge
 *
 * So, $this->paymentBridge->getOrder() must return an object
 */
$this->paymentEventDispatcher->notifyPaymentOrderLoad($this->paymentBridge,
↪$paymentMethod);
```

Exceptions

PaymentCore provides a number of Exceptions to be used by the platforms. It is important to unify certain behaviors using transparently payment platform.

PaymentAmountsNotMatchException

This exception must be thrown when the value of the payment goes through form, is validated and is not equal to the real value of the payment.

PaymentOrderNotFoundException

Launched the first event of the kernel, as explained in *Order load event*, PaymentBridge order must have a private variable in `order`. This implies that the `getOrder()` should return an object. This exception must be thrown if this method returns `null`.

PaymentExtraDataFieldNotDefinedException

As explained in *Extra Data fields* may have platforms that require extra fields. You can throw this exception if one of the camps is not found and is required.

PaymentException

Any exceptions regarding payment methods `PaymentException` extends so you can try a transparent any exception concerning PaymentCore.

Kernel Events

Order load

This event recieves as parameter an instance of

`Mmoreram\PaymentCoreBundle\Event\PaymentOrderLoadEvent` with these methods.

- `$event->getPaymentBridge` returns the implementation of `PaymentBridgeInterface` needed by PaymentCore.
- `$event->getPaymentMethod` returns the implementation of `PaymentMethodInterface` implemented by Method Platform.

```
services:
    my_event_listener:
        class: AcmeBundle\EventListener\MyEventListener
        tags:
          - { name: kernel.event_listener, event: payment.order.load, method:␣
↪onOrderLoad }
```

Order created

This event recieves as parameter an instance of

`Mmoreram\PaymentCoreBundle\Event\PaymentOrderCreatedEvent` with thow methods.

- `$event->getPaymentBridge` returns the implementation of `PaymentBridgeInterface` needed by PaymentCore.
- `$event->getPaymentMethod` returns the implementation of `PaymentMethodInterface` implemented by Method Platform.

```
services:
    my_event_listener:
        class: AcmeBundle\EventListener\MyEventListener
        tags:
          - { name: kernel.event_listener, event: payment.order.created, method:␣
↪onOrderCreated }
```

Order done

This event recieves as parameter an instance of

`Mmoreram\PaymentCoreBundle\Event\PaymentOrderDone` with thow methods.

- `$event->getPaymentBridge` returns the implementation of `PaymentBridgeInterface` needed by PaymentCore.
- `$event->getPaymentMethod` returns the implementation of `PaymentMethodInterface` implemented by Method Platform.

```
services:
    my_event_listener:
        class: AcmeBundle\EventListener\MyEventListener
        tags:
          - { name: kernel.event_listener, event: payment.order.load, method:␣
↪onOrderDone }
```

Order success

This event recieves as parameter an instance of

`Mmoreram\PaymentCoreBundle\Event\PaymentOrderSuccessEvent` with thow methods.

- `$event->getPaymentBridge` returns the implementation of `PaymentBridgeInterface` needed by PaymentCore.
- `$event->getPaymentMethod` returns the implementation of `PaymentMethodInterface` implemented by Method Platform.

```
services:
    my_event_listener:
        class: AcmeBundle\EventListener\MyEventListener
        tags:
          - { name: kernel.event_listener, event: payment.order.load, method:␣
↪onOrderSuccess }
```

Order fail

This event recieves as parameter an instance of

`Mmoreram\PaymentCoreBundle\Event\PaymentOrderFailEvent` with thow methods.

- `$event->getPaymentBridge` returns the implementation of `PaymentBridgeInterface` needed by PaymentCore.
- `$event->getPaymentMethod` returns the implementation of `PaymentMethodInterface` implemented by Method Platform.

```
services:
    my_event_listener:
        class: AcmeBundle\EventListener\MyEventListener
        tags:
          - { name: kernel.event_listener, event: payment.order.load, method:␣
↪onOrderFail}
```

Contribute

All code is Symfony2 Code formatted, so every pull request must validate phpcs standards. You should read Symfony2 coding standards and install this CodeSniffer to check all code is validated.

There is also a policy for contributing to this project. All pull request must be all explained step by step, to make us more understandable and easier to merge pull request. All new features must be tested with PHPUnit.

If you'd like to contribute, please read the Contributing Code part of the documentation. If you're submitting a pull request, please follow the guidelines in the Submitting a Patch section and use the Pull Request Template.

When contributing with PaymentCoreBundle, you can contact yuhu@mmoreram.com to let us know about your contribution in this amazing project.

Contributors

- Marc Morera (Main developer) - [@mmoreram](http://github.com/mmoreram)
- Denys Pasishnyi - [@dpcat237](http://github.com/dpcat237)
- Gonzalo Miguez - [@mrzard](http://github.com/mrzard)
- Aldo Chiecchia - [@alch](http://github.com/alch)
- Santi Castells - [@scastells](http://github.com/scastells)
- Contributors

TO DO