SAREhub

Wydanie 0.4.7

SARE SA

01 lut 2018

Spis treści

ROZDZIAŁ 1

Rurociągi

1.1 Tworzenie rurociągu

Rurociągi w kliencie PHP służą do implementowania procesorów, które wykonują procesy wymagane do wykonania ustalonych funkcjonalności.

```
<?php
  Pipeline::start()
    ->add(new Processor1())
    ->add(new Processor2());
 ?>
```

W pokazany powyżej sposób tworzymy nowy rurociąg, do którego za pomocą metody **add()** wrzucamy nowe procesory.

1.2 Uruchamianie rurociągu

Aby wywołać wykonanie rurociągu należy wywołać metodę **process()** na instancji klasy Pipeline.

```
<?php
  Pipeline::start()
  ->add(new Processor1())
  ->process();
 ?>
```

W taki sposób akcje, które zostały zdefiniowane w procesorze o nazwie Processor1 zostaną wykonane poprzez wywołanie metody process() na obiekcie klasy Pipeline.

ROZDZIAŁ 2

Procesory

2.1 Czym jest procesor?

Procesor jest klasą, która implementuje metody wywołujące procesy zaimplementowane przez nas. Przykładowo wysłanie wiadomości będzie rozdzielone na kilka procesów, takich jak: zbudowanie wiadomości, definiowanie miejsca wysyłki, wysyłka wiadomości. Procesy te wywołane zostaną w metodzie **process**(), która jest zaimplementowana w interfejsie procesora.

2.2 Tworzenie procesora

Procesor tworzymy poprzez utworzenie nowej klasy z sufiksem Processor, przykładowo *SendMessageProcessor*. Klasa ta implementuje interfejs o nazwie *Processor*.

```
<?php
  class SendMessageProcessor implements Processor {
      public function process(Exchange $exchange) {
        //...
      }
  }
 ?>
```

Powyższy kod pokazuje przykładową zawartość klasy procesora.

2.3 Uruchamianie procesora

Stworzoną klase procesora wywołujemy poprzez wrzucenie jej do rurociągu za pomocą metody **add**().

```
<?php
  Pipeline::start()
  ->add(new SendMessageProcessor())
```

```
  ->process();
 ?>
```

2.4 Jak przetworzyć treść wiadomości do formatu JSON?

Aby przetworzyć treść wiadomości najlepiej będzie skorzystać z gotowych procesorów zaimplementowanych w kliencie PHP. W tym wypadku skorzystamy z procesora o nazwie *MarshalProcessor*, który przetwarza wiadomości na różne formaty. Przypuśćmy, że wrzuciliśmy już gotową wiadomość za pomocą procesora *SendMessageProcessor* do rurociągu i mamy aktualnie następujący kod.

```
<?php
  Pipeline::start()
  ->add(new SendMessageProcessor())
  ->process();
 ?>
```

Teraz, by przekonwertować wiadomość do formatu JSON wystarczy dodać nową instancje procesora MarshalProcessor ustawiając typ konwertowania na dany format i wiadomość, która przechodzi przez rurociąg zostanie przetworzona na żądany typ. W praktyce wygląda to mniejwięcej tak:

```
<?php
  Pipeline::start()
  ->add(new SendMessageProcessor())
  ->add(MarshalProcessor::withDataFormat(new JsonDataFormat()))
  ->process();
 ?>
```

Wiadomość wysyłana przez procesor *SendMessageProcessor* trafia do kolejnego procesora, który formatuje ją na sprecyzowany klasą o nazwie *JsonDataFormat* format.

ROZDZIAŁ 3

Kontekst

3.1 Czym jest kontekst?

Kontekst jest zbiorem wartości i ustawień dla systemu. Możemy w nim przekazywać usługi pomiędzy klasami za pomocą klasy typu *Provider*, bądź ustawić globalne zmienne.

3.2 Inicjalizacja kontekstu

Kontekst tworzymy za pomocą instancji klasy implementującej interfejs *ClientContext*. W kliencie gotowa jest klasa o nazwie *BasicClientContext*, która implementuje wyżej wymieniony interfejs.

```
<?php
  BasicClientContext::newInstance()
    ->setProperty('mysqlHost', 'localhost');
  ?>
```

Tak utworzony kontekst możemy przekazywać pomiędzy klasami, dzięki temu w każdej z klas będziemy mieli dostęp do danych MySQL.

3.3 Jak przekazać serwis do klasy?

Aby przekazać serwis, który odpowiada za różne działania żądane w stworzonej klasie, należy w niej czytać kontekst. Przykładowo tworzymy właściwość przechowującą klasy implementujące ClientContext oraz za pomocą DependencyInjection wstrzykujemy kontekst do klasy.

```
<?php
  class TestClass {
    /**
    * @var ClientContext
    */
```

```
    private $context;

    public function __construct(ClientContext $c) {
      $this->context = $c;
    }
  }
 ?>
```

Tak przekazany kontekst możemy teraz odczytać za pomocą odwołania się do właściwości *$context*. Przypuśćmy na ten moment, że posiadamy usługę, która posiada możliwość dostępu do wszystkich obiektów w bazie danych oraz zarządzania nimi. Niech będzie to klasa *DatabaseManagerService*. Klase tą razem z jej zapisanymi właściwościami możemy przekazać do drugiej klasy za pomocą kontekstu, przypisując do niego serwis metodą **addService()**.

```
<?php
  class AppContextProvider implements ContextProvider {

    public function register(ClientContext $c) {
      $c->addService(DatabaseManagerService::ENTRY, new DatabaseManagerService(
↪'localhost', 'root', 'testpasswd', 'db'));
    }
  }
 ?>
```

W taki sposób dodaliśmy do kontekstu serwis, aby go pobrać możemy użyć metody **getProperty()**.

```
<?php
  class TestClass {
    /**
    * @var ClientContext
    */
    private $context;

    public function __construct(ClientContext $c) {
      $this->context = $c;
    }

    private function updateProfile($profile) {
      /** @var DatabaseManagerService $manager */
      $manager = $this->context->getProperty(DatabaseManagerService::ENTRY);

      $manager
        ->getProfile($profile->getKey())
        ->setData($profile)
        ->update();
    }
  }
 ?>
```

ROZDZIAŁ 4

Wymiana

4.1 Czym jest wymiana?

Jest to miejsce, w którym aktualnie są przechowywane wiadomości przepływające przez rurociąg. To właśnie wiadomości z wymiany są przetwarzane przez procesy z procesorów.

4.2 Tworzenie wymiany

Wymiane tworzymy poprzez stworzenie nowej instancji klasy *BasicExchange*. Następnie ustawiamy różne wartości wejścia i wyjścia(metody **setIn()** oraz **setOut()**), w których wrzucamy instancje klasy implementującej interfejs wiadomości, przykładowo *BasicMessage*.

```
<?php
BasicExchange::newInstance()
  ->setIn(BasicMessage::newInstance());
 ?>
```

W taki sposób wrzuciliśmy nową wiadomość do wymiany, która odbywa się w rurociągu. Podczas wymiany wiadomość przejdzie przez różne procesy, które zostaną zaimplementowane, oraz dojdzie do miejsca docelowego.

ROZDZIAŁ 5

Wiadomości

5.1 Czym jest wiadomość?

Wiadomosć jest obiektem, który przechowuje dane przekazywane do różnych miejsc. Może być przekazywana w różnych formatach, jest wrzucana do wymiany.

5.2 Tworzenie wiadomości

Kreacja wiadomości odbywa się poprzez stworzenie nowego obiektu klasy implementującej interfejs wiadomości, przykładową klasą istniejącą już w kliencie jest klasa *BasicMessage*.

```
<?php
BasicMessage::newInstance()
  ->setBody('Przykładowa wiadomość.');
 ?>
```

W taki sposób stworzyliśmy wiadomość z ciałem zawierającym tekst *Przykładowa wiadomość.*. Teraz należy w odpowiednim momencie pobrać wiadomość z wymiany, aby ją wyświetlić bądź przekazać dalej.

ROZDZIAŁ 6

Użytkownik

6.1 Jak działa użytkownik w kliencie?

Użytkownik w kliencie unifikuje przesyłane dane w wiadomościach, pozwala ustawić w wiadomości identyfikatory typu:

- cookie - identyfikator przechowywany w danych przeglądarki.
- id - identyfikator numeryczny klienta.
- email - identyfikator zawierający adres e-mail.
- mobile - identyfikator zawierający numer telefonu.

Typy identyfikatorów są przechowywane w interfejsie UserKeys jako stałe.

6.2 Tworzenie użytkownika

Do stworzenia użytkownika potrzebna będzie kreacja nowej instancji klasy User, następnie należy przypisać instancji odpowiednie atrybuty. Wygląda to w następujący sposób:

```
<?php
  $user = new User([
    UserKeys::ID => 123,
    UserKeys::COOKIE => 14321123416123413215,
    UserKeys::EMAIL => 'jan.kowalski@sarehub.com'
  ]);
 ?>
```

ROZDZIAŁ 7

Zdarzenia

7.1 Tworzenie zdarzenia

Do stworzenia zdarzenia w kliencie należy skorzystać z klasy implementującej interfejs *Event*. Zdarzenie posiada swoje właściwości, przypisanego użytkownika, czas wykonania zdarzenia oraz jego nazwę. Działa to w sposób pokazany poniżej:

```
<?php
  BasicEvent::newInstance('message_send')
    ->withUser(new User([UserKeys::COOKIE => 134214123412341]))
    ->withTime(time())
    ->withProperties([
      'message_id' => 2
    ])
 ?>
```

Tak stworzone zdarzenie można posłać do brokera wiadomości, który przekaże informacje o zdarzeniu dalej i zainicjuje wykonanie różnych procesów.

ROZDZIAŁ 8

Przepisy

8.1 Jak wrzucić zdarzenie do wymiany?

Na początku aby wrzucić zdarzenie do wymiany musimy zdecydować, co w zdarzeniu ma się znajdować. Załóżmy, że chcemy wysłać zdarzenie, które poinformuje system o wysłaniu wiadomości, którą system zażądał wysłać.

```
<?php
  private function buildExchange(SystemMessage $message): Exchange {
              return BasicExchange::newInstance()
                ->setIn(BasicMessage::newInstance()
                      ->setBody(BasicEvent::newInstanceOf('message_sent')
                        ->withProperties([
                              'id' => $message->getId()
                        ])
                        ->withUser(new User([UserKeys::ID => $message->getAuthor()->
↪getId()]))
                        ->withTime(time())));
      }
 ?>
```

Mamy już zdeklarowaną wymianę razem ze zdarzeniem, teraz musimy pchnąć ją do procesora, aby ten przesłał informację o tym do systemu.

```
<?php
    class SendEventToSystemProcessor implements Processor {
      public function process(Exchange $exchange) {
          $message = $exchange->getIn()->getBody();
          $this->sendMessage($message);
      }

      private function sendMessage(Message $message) {
        //Wysyłanie wiadomości do brokera.
      }
    }
 ?>
```

Ostatnim etapem jest utworzenie rurociągu, który przetworzy nasze działanie.

```
<?php
    class ModulePipelineFactory {
      /**
      * @var Exchange
      */
      private $exchange;

      public function create() {
        $this->exchange = $this->buildExchange($messageToSet);
        return $this->process();
      }

      private function process() {
        return Pipeline::start()
          ->add(new SendEventToSystemProcessor())
          ->process($this->exchange)
      }

      private function buildExchange(SystemMessage $message): Exchange {
              return BasicExchange::newInstance()
                ->setIn(BasicMessage::newInstance()
                      ->setBody(BasicEvent::newInstanceOf('message_sent')
                        ->withProperties([
                              'id' => $message->getId()
                        ])
                        ->withUser(new User([UserKeys::ID => $message->getAuthor()->
↪getId()]))
                        ->withTime(time())));
      }
    }
 ?>
```

W taki sposób można zaimplementować wysyłanie zdarzeń, do zmiennej **$messageToSet** należy przypisać odpowiednie dane, które mogą być przekazane na przykład w klasie typu **Provider**.