Technologia i rozwiązania

Symfony2
Rozbudowa frameworka

Odkryj nowe możliwości Symfony2!

Sébastien Armand

Helion

[PACKT] PUBLISHING open source community experience distilled

Tytuł oryginału: Extending Symfony 2 Web Application Framework

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-0294-5

First published in the English language under the title „Extending Symfony 2 Web Application Framework".

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/sym2rf
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/sym2rf.zip

Printed in Poland.

Spis treści

O autorze 5

O recenzentach 7

Wstęp 9

Zawartość książki 9
Co jest potrzebne? 10
Dla kogo jest ta książka? 10
Konwencje 11
Pomoc 11

Rozdział 1. Usługi i procedury nasłuchowe 13

Usługi 13
Usługa geolokalizacji 14
Testowanie usług i testowanie przy użyciu usług 19
Znakowanie usług 21
Procedury nasłuchujące 25
Aktualizowanie preferencji użytkownika przy użyciu własnych zdarzeń 27
Poprawianie wydajności 30
Podsumowanie 32

Rozdział 2. Polecenia i szablony 33

Polecenia 33
Początkowa sytuacja 33
Zmienianie rozmiaru obrazów użytkowników 34
Testowanie polecenia 37
Polecenia jako interfejs do usług 38
Twig 40
Zarządzanie skryptami 41
Testowanie rozszerzenia Twig 43
Filtr różnicy czasowej 44
Podsumowanie 45

Rozdział 3. Formularze **47**

Element wejściowy dla współrzędnych geograficznych **47**
Podstawowa konfiguracja 49
Używanie mapy 51
Przekształcanie danych **54**
Formularze wykorzystujące dane użytkowników **56**
O krok dalej **57**
Początkowa konfiguracja 58
Dodawanie i usuwanie pól 60
Podsumowanie **62**

Rozdział 4. Bezpieczeństwo **63**

Uwierzytelnianie **63**
Proste uwierzytelnianie OAuth poprzez GitHub 64
Autoryzacja **74**
Votery 75
Adnotacje 80
Zabezpieczanie API — przykład **85**
Podsumowanie **87**

Rozdział 5. Doctrine **89**

Tworzenie własnych typów danych **89**
Miejsce przebywania użytkowników i miejsca spotkań 89
Testowanie 92
Własne funkcje DQL **93**
Kontrola wersji **97**
Ustawianie wersji wszystkich jednostek 99
Używanie i aktualizowanie wersji 100
Testowanie 101
Tworzenie filtra Doctrine **103**
Podsumowanie **106**

Rozdział 6. Udostępnianie własnych rozszerzeń innym programistom **107**

Tworzenie pakietu **107**
Udostępnianie konfiguracji 110
Przygotowanie do udostępnienia pakietu **116**
Badania 116
Dokumentacja 116
Testowanie 116
Dystrybucja i licencjonowanie 118
Czy to jest tylko pakiet? 119
Podsumowanie **120**

Skorowidz **121**

1

ROZDZIAŁ

Usługi i procedury nasłuchowe

W rozdziale tym znajduje się opis podstawowych wiadomości na temat systemu Symfony2. Najważniejszym pojęciem jest **usługa** (ang. *service*). W istocie większa część samego systemu jest wielkim zbiorem gotowych do używania usług. Na przykład po zainstalowaniu systemu można przejść w konsoli do katalogu głównego projektu i wpisać polecenie `php app/console container:debug`, aby wyświetlić listę wszystkich aktualnie zdefiniowanych w aplikacji usług. Jeśli to zrobisz, dowiesz się, że nawet jeszcze przed rozpoczęciem pracy masz do dyspozycji prawie 200 usług. Polecenie `php app/console container:debug <nazwa_usługi>` zwraca informacje o wybranej usłudze; przyda się ono wielokrotnie w trakcie studiowania tej książki.

Usługi

Usługa jest konkretnym egzemplarzem jakiejś klasy. Gdy programista używa, powiedzmy, `doctrine`, np. `$this->get('doctrine');` w kontrolerze, znaczy to, że korzysta z usługi. Ta usługa jest egzemplarzem klasy `Doctrine EntityManager`, którego nigdy nie trzeba tworzyć samodzielnie. Kod potrzebny do jego utworzenia jest dość skomplikowany, ponieważ wymaga połączenia z bazą danych, pewnych parametrów konfiguracyjnych itd. Gdyby ta usługa nie była już zdefiniowana, trzeba by tworzyć takie egzemplarze samodzielnie. Gdyby zaszła konieczność zrobienia tego w każdym kontrolerze, kod aplikacji stałby się zagmatwany i trudny w obsłudze.

Oto kilka z domyślnych usług dostępnych w Symfony2:

- czytnik adnotacji,
- Assetic — biblioteka do zarządzania zasobami,

- dyspozytor zdarzeń,
- fabryka widżetów formularza i formularzy,
- jądro i składnik HttpKernel Symfony2,
- monolog — biblioteka obsługi dzienników,
- ruter,
- Twig — silnik szablonów.

W systemie Symfony2 bardzo łatwo tworzy się nowe rozszerzenia. Jeśli Twój kontroler bardzo się rozrósł i trudno nad nim zapanować, dobrym sposobem jego poprawienia i uproszczenia jest przesunięcie części kodu do usług. Większość usług to obiekty singletonowe, czyli mogące występować tylko w pojedynczym egzemplarzu.

Usługa geolokalizacji

Wyobraź sobie aplikację tworzącą listy zdarzeń, które nazwiemy „spotkaniami". Kontroler umożliwia nam pobranie najpierw adresu IP bieżącego użytkownika, sprawdzenie z wykorzystaniem tego IP lokalizacji tego użytkownika oraz wyświetlenie spotkań w promieniu 50 kilometrów. Aktualnie cały kod znajduje się w kontrolerze. Na razie jeszcze kontroler ten nie jest zbyt długi — zawiera jedną metodę i cała klasa zajmuje jakieś 50 wierszy kodu. Ale z czasem dodamy więcej kodu, aby na przykład móc wyświetlać tylko ulubione spotkania użytkownika albo takie, w których użytkownik brał udział najczęściej. Gdy połączy się te wszystkie informacje i doda skomplikowane obliczenia mające na celu znalezienie najodpowiedniejszych spotkań dla danego użytkownika, kod może rozrosnąć się do niebotycznych rozmiarów!

Ten prosty problem można rozwiązać na kilka sposobów. Logikę geokodowania można na razie przenieść do osobnej metody. Będzie to dobre tymczasowe posunięcie, ale lepiej myśleć przyszłościowo i przenieść część logiki do usług, do których należy. Aktualnie nasz kod wygląda tak:

```
use Geocoder\HttpAdapter\CurlHttpAdapter;
use Geocoder\Geocoder;
use Geocoder\Provider\FreeGeoIpProvider;

public function indexAction()
  {
```

Narzędzia do geokodowania (oparte na doskonałej bibliotece geokodowania — *http://geocoder-php.org/*) zainicjujemy przy użyciu następującego kodu:

```
$adapter = new CurlHttpAdapter();
$geocoder = new Geocoder();
$geocoder->registerProviders(array(
  new FreeGeoIpProvider($adapter),
));
```

Pobieramy adres IP użytkownika:

```
$ip = $this->get('request')->getClientIp();
// Można też użyć domyślnego.
if ($ip == '127.0.0.1') {
  $ip = '114.247.144.250';
}
```

Pobieramy współrzędne i dostosowujemy je przy użyciu poniższego kodu, aby tworzyły mniej więcej kwadrat o boku 50 km:

```
$result = $geocoder->geocode($ip);
$lat = $result->getLatitude();
$long = $result->getLongitude();
$lat_max = $lat + 0.25;  // około 25 km
$lat_min = $lat - 0.25;
$long_max = $long + 0.3; // około 25 km
$long_min = $long - 0.3;
```

Na podstawie tych wszystkich informacji tworzymy zapytanie:

```
$em = $this->getDoctrine()->getManager();
$qb = $em->createQueryBuilder();
$qb->select('e')
    ->from('KhepinBookBundle:Meetup', 'e')
    ->where('e.latitude < :lat_max')
    ->andWhere('e.latitude > :lat_min')
    ->andWhere('e.longitude < :long_max')
    ->andWhere('e.longitude > :long_min')
    ->setParameters([
      'lat_max' => $lat_max,
      'lat_min' => $lat_min,
      'long_max' => $long_max,
      'long_min' => $long_min
    ]);
```

Pobieramy wyniki i przekazujemy je do szablonu:

```
$meetups = $qb->getQuery()->execute();
return ['ip' => $ip, 'result' => $result,
    'meetups' => $meetups];
}
```

Chcemy się pozbyć inicjacji geokodowania. Najlepiej, żeby wszystko to odbywało się automatycznie, a dostęp do geokodera odbywał się za pomocą instrukcji `$this->get('geocoder');`.

Skąd pobrać przykłady kodu?

Pliki z przykładami kodu źródłowego można pobrać z serwera FTP wydawnictwa Helion pod adresem *ftp://ftp.helion.pl/przyklady/sym2rf.zip*.

Usługi można definiować bezpośrednio w pliku *config.yml* systemu Symfony pod kluczem `services`, jak pokazano poniżej:

```
services:
  geocoder:
    class: Geocoder\Geocoder
```

To wszystko! Zdefiniowaliśmy usługę, która jest teraz dostępna we wszystkich naszych kontrolerach. Teraz nasz kod wygląda tak:

```
// Tworzy klasę geokodowania.
$adapter = new \Geocoder\HttpAdapter\CurlHttpAdapter();
$geocoder = $this->get('geocoder');
$geocoder->registerProviders(array(
    new \Geocoder\Provider\FreeGeoIpProvider($adapter),
));
```

Już widzę, jak przewracasz oczami i stwierdzasz, że to niewiele pomaga. Jest tak, ponieważ inicjacja geokodera jest nieco bardziej skomplikowana niż zwykłe wywołanie `new \Geocoder\Geocoder()`. Konieczne jest utworzenie obiektu innej klasy i przekazanie go jako parametru do metody. Dobra wiadomość jest taka, że wszystko to można zrobić w definicji usługi. Wystarczy tylko ją zmodyfikować w następujący sposób:

```
services:
    # Definiuje klasę adaptacyjną.
    geocoder_adapter:
        class: Geocoder\HttpAdapter\CurlHttpAdapter
        public: false
    # Definiuje klasę dostawczą.
    geocoder_provider:
        class: Geocoder\Provider\FreeGeoIpProvider
        public: false
        # Klasie dostawczej jest przekazywany adapter jako argument.
        arguments: [@geocoder_adapter]
    geocoder:
        class: Geocoder\Geocoder
        # Po inicjacji wywołujemy na geokoderze metodę, aby ustawić odpowiednie parametry.
        calls:
            - [registerProviders, [[@geocoder_provider]]]
```

Ten kod jest już trochę dłuższy, ale to jest jedyne miejsce, w którym musimy go napisać. Warto zwrócić uwagę na parę rzeczy:

- W rzeczywistości zdefiniowaliśmy trzy usługi, ponieważ nasz geokoder wymaga egzemplarzy dwóch innych klas.
- Aby przekazać referencję do usługi jako argument do innej usługi, użyliśmy składni `@+nazwa_usługi`.

- Nie musimy się ograniczać tylko do definicji `new Class($argument);`. Możemy też wywołać metodę na klasie po utworzeniu jej egzemplarza. Istnieje nawet możliwość bezpośredniego ustawiania właściwości, jeśli są publiczne.
- Dwie pierwsze usługi oznaczyliśmy jako prywatne, co znaczy, że nie będą dostępne w kontrolerach. Mogą natomiast być wstrzykiwane przez **kontener wstrzykiwania zależności** (ang. *dependency injection container* — DIC) do innych usług.

Teraz nasz kod wygląda tak:

```
// Pobiera adres IP użytkownika.
$ip = $this->get('request')->getClientIp();

// Albo używa domyślnego.
if ($ip == '127.0.0.1') {
    $ip = '114.247.144.250';
}
// Sprawdza współrzędne użytkownika.
$result = $this->get('geocoder')->geocode($ip);
$lat = $result->getLatitude();
// ... Reszta kodu pozostaje bez zmian.
```

W tym przypadku kontrolery rozszerzają klasę `BaseController`, która ma dostęp do DIC, ponieważ implementuje interfejs `ContainerAware`. Wszystkie wywołania `$this->get('nazwa_usługi')` są przekazywane kontenerowi, który konstruuje (w razie potrzeby) i zwraca usługę.

Posuniemy się jeszcze dalej i zdefiniujemy własną klasę, która bezpośrednio będzie pobierać adres IP użytkownika oraz zwracać tablicę maksymalnych i minimalnych długości i szerokości geograficznych. Utworzymy następującą klasę:

```
namespace Khepin\BookBundle\Geo;

use Geocoder\Geocoder;
use Symfony\Component\HttpFoundation\Request;

class UserLocator {

    protected $geocoder;

    protected $user_ip;

    public function __construct(Geocoder $geocoder, Request
      $request) {
        $this->geocoder = $geocoder;
        $this->user_ip = $request->getClientIp();
        if ($this->user_ip == '127.0.0.1') {
            $this->user_ip = '114.247.144.250';
        }
```

```
    }

    public function getUserGeoBoundaries($precision = 0.3) {
        // Sprawdza współrzędne użytkownika.
        $result = $this->geocoder->geocode($this->user_ip);
        $lat = $result->getLatitude();
        $long = $result->getLongitude();
        $lat_max = $lat + 0.25;  // około 25 km
        $lat_min = $lat - 0.25;
        $long_max = $long + 0.3; // około 25 km
        $long_min = $long - 0.3;
        return ['lat_max' => $lat_max, 'lat_min' => $lat_min,
            'long_max' => $long_max, 'long_min' => $long_min];
    }
}
```

Konstruktor tej klasy przyjmuje jako argumenty zmienne `geocoder` i `request`, a następnie klasa ta wykonuje całą pracę, którą na początku wykonywaliśmy w kontrolerze. Podobnie jak wcześniej, klasę tę zdefiniujemy jako usługę, aby była łatwo dostępna w kontrolerach:

```
# config.yml
services:
    #...
user_locator:
    class: Khepin\BookBundle\Geo\UserLocator
    scope: request
    arguments: [@geocoder, @request]
```

Zwróć uwagę na definicję zakresu w tym kodzie. DIC ma domyślnie dwa zakresy: `container` i `prototype`, do których system dodaje jeszcze trzeci, o nazwie `request`. W poniższej tabeli znajduje się opis różnic między nimi.

Zakres	Różnice
`container`	Wszystkie wywołania `$this->get('service_name')` zwracają ten sam egzemplarz usługi.
`prototype`	Wszystkie wywołania `$this->get('service_name')` zwracają nowy egzemplarz usługi.
`request`	Wszystkie wywołania `$this->get('service_name')` zwracają ten sam egzemplarz usługi w żądaniu. Symfony może mieć żądania podrzędne (np. zawierające kontroler w Twig).

Z wykonanych działań odnieśliśmy taką korzyść, że usługa samodzielnie zdobywa wszystkie potrzebne jej informacje, ale niestety staje się bezużyteczna w kontekstach, w których nie ma żądań. Gdybyśmy chcieli utworzyć polecenie pobierające wszystkie adresy IP, z którymi łączył się użytkownik, i wysyłające mu wiadomości o spotkaniach odbywających się w weekend w jego okolicy, to ten projekt uniemożliwiłby nam użycie potrzebnej do tego klasy `Khepin\BookBundle\Geo\UserLocator`.

Jak widać, domyślnie usługi znajdują się w zakresie kontenera, co znaczy, że ich egzemplarz jest tworzony tylko raz, a potem wielokrotnie używany zgodnie z zasadami wzorca projektowego Singleton. Ponadto należy zauważyć, że DIC nie tworzy wszystkich usług natychmiast, tylko na żądanie. Jeśli kod znajdujący się w innym kontrolerze nie używa usługi `user_locator`, to ani ta usługa, ani żadna z usług, od których zależy (`geocoder`, `geocoder_provider` i `geocoder_adapter`), nie zostanie utworzona.

Ponadto należy pamiętać, że konfiguracja zapisana w pliku *config.yml* jest buforowana w środowisku produkcyjnym, dzięki czemu definicja tych usług powoduje minimalny lub wręcz zerowy narzut.

Teraz nasz kontroler jest już znacznie prostszy i wygląda następująco:

```
$boundaries = $this->get('user_locator')->getUserGeoBoundaries();
// Tworzy zapytanie do bazy danych.
$em = $this->getDoctrine()->getManager();
$qb = $em->createQueryBuilder();
$qb->select('e')
    ->from('KhepinBookBundle:Meetup', 'e')
    ->where('e.latitude < :lat_max')
    ->andWhere('e.latitude > :lat_min')
    ->andWhere('e.longitude < :long_max')
    ->andWhere('e.longitude > :long_min')
    ->setParameters($boundaries);
// Pobiera informacje o interesujących spotkaniach.
$meetups = $qb->getQuery()->execute();
return ['meetups' => $meetups];
```

Najwięcej miejsca zajmuje zapytanie `Doctrine`, które łatwo można przenieść do klasy repozytorium, aby jeszcze bardziej uprościć kontroler.

Jak widać na przedstawionym przykładzie, definiowanie i tworzenie usług w Symfony2 jest dość łatwe i niezbyt kosztowne. Utworzyliśmy własną klasę `UserLocator`, zamieniliśmy ją w usługę oraz dowiedzieliśmy się, że może ona zależeć od innych naszych usług, np. `@geocoder`. Nie skończyliśmy jeszcze z usługami ani DIC, ponieważ są to podstawowe składniki prawie wszystkich technik związanych z rozszerzaniem systemu Symfony2. Będzie o nich mowa jeszcze wiele razy w tej książce i dlatego zanim przejdziemy dalej, koniecznie musimy je dobrze zrozumieć.

Testowanie usług i testowanie przy użyciu usług

Jedną z wielkich zalet umieszczania kodu w usługach jest to, że usługi są po prostu klasami PHP. Dzięki temu można je szybko testować. Nie trzeba do tego kontrolera ani DIC. Wystarczy tylko utworzyć atrapy klas `geocoder` i `request`.

W folderze *test* pakietu można utworzyć folder o nazwie *Geo*, w którym będziemy testować naszą klasę `UserLocator`. Jako że testowana będzie zwykła klasa PHP, nie trzeba używać klasy `WebTestCase`. Wystarczy nam standardowa klasa `PHPUnit_Framework_TestCase`. Nasza klasa zawiera

tylko jedną metodę geokodującą adres IP i zwracającą zbiór współrzędnych określonych z wyznaczoną precyzją. Możemy imitować działanie geokodera przez zwracanie na sztywno ustawionych liczb, dzięki czemu nie będziemy musieli wykonywać wywołań sieciowych, które spowolniłyby nasze testy. Poniżej znajduje się prosty przypadek testowy:

```
class UserLocatorTest extends PHPUnit_Framework_TestCase
{
    public function testGetBoundaries()
    {
        $geocoder = $this->getMock('Geocoder\Geocoder');
        $result = $this->getMock('Geocoder\Result\Geocoded');

        $geocoder->expects($this->any())->method('geocode')-
          >will($this->returnValue($result));
        $result->expects($this->any())->method('getLatitude')-
          >will($this->returnValue(3));
        $result->expects($this->any())->method('getLongitude')
          ->will($this->returnValue(7));

        $request = $this->getMock
           ('Symfony\Component\HttpFoundation\Request',
             ['getUserIp']);
        $locator = new UserLocator($geocoder, $request);

        $boundaries = $locator->getUserGeoBoundaries(0);

        $this->assertTrue($boundaries['lat_min'] == 3);
    }
}
```

Teraz możemy sprawdzić, czy działa nasza klasa, ale co z resztą logiki kontrolera?

Dla kontrolera możemy napisać prosty test integracyjny, aby sprawdzić, czy na wyrenderowanej stronie znajdują się informacje o jakichś spotkaniach. Ale w niektórych przypadkach podczas testowania lepiej jest nie wywoływać zewnętrznych usług ze względu na wydajność, wygodę lub po prostu brak takiej możliwości. W takiej sytuacji również można posłużyć się atrapami usług, które będą używane w kontrolerze. W naszych testach musimy to zrobić tak:

```
public function testIndexMock()
{
    $client = static::createClient();
    $locator = $this->getMockBuilder
       ('Khepin\BookBundle\Geo\UserLocator')
        ->disableOriginalConstructor()->getMock();
    $boundaries = ["lat_max" => 40.2289, "lat_min" => 39.6289,
        "long_max" => 116.6883, "long_min" => 116.0883];
    $locator->expects($this->any())->method
         ('getUserGeoBoundaries')->will($this-
          >returnValue($boundaries));
```

```
        $client->getContainer()->set('user_locator', $locator);
        $crawler = $client->request('GET', '/');
        // Sprawdza, czy strona zawiera oczekiwane informacje o spotkaniach.
    }
```

W kodzie tym utworzyliśmy atrapę klasy `UserLocator`, która zawsze zwraca te same współrzędne. Dzięki temu mamy większą kontrolę nad tym, co testujemy, i nie musimy długo czekać na wywołanie serwera geolokacyjnego.

Znakowanie usług

Zapewne podczas używania systemu Symfony spotkałeś się już z oznakowanymi usługami, np. przy definiowaniu własnych widżetów formularza albo voterów zabezpieczeń. Oznakowanymi usługami są też procedury nasłuchu zdarzeń, o których będzie mowa w drugiej części tego rozdziału.

W poprzednich przykładach utworzyliśmy usługę `user_locator`, której działanie zależy od usługi geokodowania. Ale użytkownika można zlokalizować na wiele sposobów. Można posłużyć się danymi adresowymi z profilu, co jest szybszą i dokładniejszą metodą niż sprawdzanie według adresu IP. Można też użyć różnych dostawców internetowych, takich jak FreeGeoIp, co zrobiliśmy w poprzednim kodzie, albo utrzymywać lokalną bazę danych `geoip`. Można nawet wszystkie te techniki zaimplementować w jednej aplikacji i wypróbowywać je jedną po drugiej, zaczynając od najbardziej dokładnej.

Interfejs dla tego nowego typu geokodera zdefiniujemy następująco:

```
namespace Khepin\BookBundle\Geo;

interface Geocoder
{
    public function getAccuracy();

    public function geocode($ip);
}
```

Następnie zdefiniujemy dwa geokodery przy użyciu poniższego kodu. Pierwszy z nich opakowuje istniejący geokoder w nową klasę implementującą nasz interfejs `Geocoder`:

```
namespace Khepin\BookBundle\Geo;
use Geocoder\Geocoder as IpGeocoder;

class FreeGeoIpGeocoder implements Geocoder
{
    public function __construct(IpGeocoder $geocoder)
    {
        $this->geocoder = $geocoder;
    }
```

```
    public function geocode($ip)
    {
        return $this->geocoder->geocode($ip);
    }

    public function getAccuracy()
    {
        return 100;
    }
}
```

Pierwszy typ geokodera jest skonfigurowany następująco:

```
freegeoip_geocoder:
    class: Khepin\BookBundle\Geo\FreeGeoIpGeocoder
    arguments: [@geocoder]
```

Drugi geokoder za każdym razem zwraca losową lokalizację:

```
namespace Khepin\BookBundle\Geo;

class RandomLocationGeocoder implements Geocoder
{
    public function geocode($ip)
    {
        return new Result();
    }

    public function getAccuracy()
    {
        return 0;
    }
}

class Result
{
    public function getLatitude()
    {
        return rand(-85, 85);
    }

    public function getLongitude()
    {
        return rand(-180, 180);
    }

    public function getCountryCode()
    {
        return 'CN';
    }
}
```

Konfiguracja drugiego geokodera wygląda tak:

```
random_geocoder:
    class: Khepin\BookBundle\Geo\RandomLocationGeocoder
```

Jeśli zmienimy konfigurację naszej usługi `user_locator` tak, aby przestawić ją na używanie jednego z tych geokoderów, wszystko nam zadziała. Ale my chcemy, aby nasza usługa bez żadnych zmian w jej kodzie mogła używać wszystkich dostępnych metod oraz wybrać najbardziej precyzyjną z nich, nawet gdy zostaną dodane nowe.

Oznaczymy nasze usługi przez dodanie znaczników w ich konfiguracjach:

```
freegeoip_geocoder:
    class: Khepin\BookBundle\Geo\FreeGeoIpGeocoder
    arguments: [@geocoder]
    tags:
        - { name: khepin_book.geocoder }
random_geocoder:
    class: Khepin\BookBundle\Geo\RandomLocationGeocoder
    tags:
        - { name: khepin_book.geocoder }
```

Nie możemy ich wszystkich przekazać bezpośrednio w konstruktorze klasy, więc dodamy do klasy `UserLocator` metodę `addGeocoder`:

```
class UserLocator
{    protected $geocoders = [];

    protected $user_ip;

    // Stąd usunięto geokoder.
    public function __construct(Request $request)
    {
        $this->user_ip = $request->getClientIp();
    }

    public function addGeocoder(Geocoder $geocoder)
    {
        $this->geocoders[] = $geocoder;
    }

    // Wybiera najodpowiedniejszy geokoder.
    public function getBestGeocoder(){/* ... */}

    // ...
}
```

Nie można poinformować DIC o chęci dodania oznakowanych usług tylko przez konfigurację. Robi się to w czasie działania kompilatora — podczas kompilacji DIC.

W przebiegach kompilatora można dynamicznie modyfikować definicje usług. Można to wykorzystać dla usług oznakowanych oraz do tworzenia pakietów włączających dodatkowe funkcje, gdy jakiś inny pakiet również jest obecny i skonfigurowany. Oto przykład wykorzystania przebiegu kompilatora:

```
namespace Khepin\BookBundle\DependencyInjection\Compiler;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler
  \CompilerPassInterface;
use Symfony\Component\DependencyInjection\Reference;

class UserLocatorPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        if (!$container->hasDefinition('khepin_book.user_locator'))
          {
              return;
          }

          $service_definition = $container->getDefinition
             ('khepin_book.user_locator');
          $tagged = $container->findTaggedServiceIds
           ('khepin_book.geocoder');

          foreach ($tagged as $id => $attrs) {
              $service_definition->addMethodCall(
                  'addGeocoder',
                  [new Reference($id)]
              );
          }
     }
}
```

Po potwierdzeniu, że usługa `user_locator` (tu przemianowana na `khepin_book.user_locator`) istnieje, wyszukujemy wszystkie usługi z odpowiednim znacznikiem i modyfikujemy definicję usługi `khepin_book.user_locator` w taki sposób, aby je ładowała.

Można zdefiniować atrybuty znacznika. Dzięki temu moglibyśmy na przykład zapisać dokładność każdego geokodera w jego konfiguracji, a następnie w przebiegu kompilatora lokalizatorowi użytkownika dostarczyć najprecyzyjniejszy dekoder:

```
tags:
  - { name: khepin_book.geocoder, accuracy: 69 }
```

Gdy programista zdefiniuje konfigurację YAML dla usług, Symfony na podstawie tych informacji wewnętrznie tworzy definicje usług. Dzięki dodaniu przebiegu kompilatora możemy modyfikować te definicje w sposób dynamiczny. Definicje usług są następnie buforowane, aby nie trzeba było ponownie kompilować kontenera.

Procedury nasłuchujące

Procedury nasłuchujące realizują implementację wzorca projektowego Obserwator. We wzorcu tym wybrany fragment kodu nie próbuje rozpocząć wykonywania całego kodu, który powinien zostać wykonany w danym momencie. Zamiast tego powiadamia swoich obserwatorów, że doszedł do pewnego punktu wykonywania, i mogą oni przejąć kontrolę, jeśli jest taka potrzeba.

W Symfony wzorzec Obserwator jest realizowany przez zdarzenia. Każda klasa i funkcja może wyzwolić zdarzenie, gdy tylko uzna to za stosowne. Samo zdarzenie może być zdefiniowane w klasie. Dzięki temu można przekazać więcej informacji do obserwującego je kodu. System także zgłasza zdarzenia w różnych momentach obsługi żądań. Są to:

- `kernel.request` — to zdarzenie ma miejsce przed dotarciem do kontrolera. Jest używane wewnętrznie do zapełniania danymi obiektu `request`.
- `kernel.controller` — to zdarzenie ma miejsce bezpośrednio przed uruchomieniem kontrolera. Można je wykorzystać w celu zmiany kontrolera, który jest aktualnie wykonywany.
- `kernel.view` — to zdarzenie ma miejsce po wykonaniu kontrolera, jeśli kontroler ten nie zwrócił obiektu `response`. Można je wykorzystać do zlecenia domyślnej obsługi renderowania widoku przez Twig.
- `kernel.response` — to zdarzenie ma miejsce przed wysłaniem odpowiedzi. Można je wykorzystać do zmodyfikowania odpowiedzi przed jej wysłaniem.
- `kernel.terminate` — to zdarzenie ma miejsce po wysłaniu odpowiedzi. Można je wykorzystać do wykonania czasochłonnych operacji, które nie muszą generować odpowiedzi.
- `kernel.exception` — to zdarzenie ma miejsce, gdy system przechwyci nieobsłużony wyjątek.

> `Doctrine` także zgłasza zdarzenia w czasie cyklu istnienia obiektu (np. przed zapisaniem lub po zapisaniu go w bazie danych), ale to całkiem osobny temat. Wszystko na temat zdarzeń cyklu istnienia obiektów `Doctrine` można znaleźć na stronie *http://doctrine-orm.readthedocs.org/en/latest/reference/events.html#reference-events-lifecycle-events*.

Zdarzenia są niezwykle przydatne i dlatego będą używane jeszcze wiele razy w różnych miejscach tej książki. Gdy udostępnia się rozszerzenia do Symfony innym programistom, **zawsze** dobrym pomysłem jest zdefiniowanie i wyzwalanie własnych zdarzeń, które mogą służyć jako własnościowe punkty rozszerzeń.

Teraz rozbudujemy przykład z poprzedniej części rozdziału, aby zobaczyć, do czego mogą przydać się procedury nasłuchujące.

W pierwszej części zbudowaliśmy stronę internetową wyświetlającą informacje o spotkaniach w okolicy miejsca przebywania użytkownika. Teraz dodatkowo sprawimy, że informacje te będą filtrowane zgodnie z preferencjami użytkownika.

Aktualizujemy schemat, aby utworzyć relację „wiele do wielu" między użytkownikami i spotkaniami:

```
// Entity/User.php
/**
 * @ORM\ManyToMany(targetEntity="Meetup", mappedBy="attendees")
 */
protected $meetups;
// Entity/Meetup.php

/**
 * @ORM\ManyToMany(targetEntity="User", inversedBy="meetups")
 */
protected $attendees;
```

W kontrolerze mamy prostą akcję pozwalającą wziąć udział w spotkaniu:

```
/**
 * @Route("/meetups/{meetup_id}/join")
 * @Template()
 */
public function joinAction($meetup_id) {
    $em = $this->getDoctrine()->getManager();
    $meetup = $em->getRepository('KhepinBookBundle:Meetup')
      ->find($meetup_id);

    $form = $this->createForm(
        new JoinMeetupType(),
        $meetup,
         ['action' => '', 'method' => 'POST']
    );
    $form->add('submit', 'submit', array('label' => 'Join'));
    $form->handleRequest($this->get('request'));

    $user = $this->get('security.context')->getToken()->getUser();

    if ($form->isValid()) {
        $meetup->addAttendee($user);
```

```
            $em->flush();
        }

        $form = $form->createView();
        return ['meetup' => $meetup, 'user' => $user,
          'form' => $form];
    }
```

Użyliśmy formularza, mimo że ta akcja jest bardzo prosta, ponieważ przesyłanie wszystkich informacji w adresie URL w celu zaktualizowania bazy danych i zarejestrowania użytkownika jako uczestnika byłoby słabym punktem, narażającym aplikację na wiele ataków, np. CSRF.

Aktualizowanie preferencji użytkownika przy użyciu własnych zdarzeń

Chcemy napisać kod generujący nową listę ulubionych spotkań użytkownika. W tym celu musimy zmienić logikę wyświetlania strony głównej. Będziemy wyświetlać nie tylko listę spotkań z pobliża miejsca przebywania użytkownika, ale dodatkowo przefiltrujemy dane według preferencji tego użytkownika. Przewidujemy, że strona główna naszej aplikacji będzie często wyświetlana, przez co wykonywanie wszystkich obliczeń przy każdym jej otwarciu może być bardzo kosztowne. Dlatego lepiej będzie utworzyć gotową listę ulubionych rodzajów spotkań, którą będziemy modyfikować, gdy użytkownik zapisze się na jakieś spotkanie lub zrezygnuje z udziału w jakimś spotkaniu. W przyszłości można też listę tę aktualizować na podstawie przeglądanych stron, nawet jeśli użytkownik nie zapisze się na dane spotkanie.

Teraz musimy zastanowić się, gdzie umieścić nasz kod. Narzuca się myśl, aby wstawić go wprost do kontrolera, chociaż nie jest to właściwe miejsce. Zadaniem kontrolera jest zapewnienie użytkownikowi zapisania się na spotkanie, i tak powinno pozostać.

Ale możemy też wywołać w kontrolerze zdarzenie, które ostrzeże wszystkich obserwatorów, że użytkownik zapisał się na spotkanie. Decyzję, co zrobić z tą informacją, pozostawimy już obserwatorom.

Aby to zdarzenie było przydatne, musi zawierać dane o użytkowniku i spotkaniu. Dlatego utworzymy prostą klasę do przechowywania tych informacji:

```
// Bundle/Event/MeetupEvent.php
namespace Khepin\BookBundle\Event;

use Symfony\Component\EventDispatcher\Event;
use Khepin\BookBundle\Entity\User;
use Khepin\BookBundle\Entity\Meetup;
```

```
class MeetupEvent extends Event
{
    protected $user;
    protected $event;

    public function __construct(User $user, Meetup $meetup) {
        $this->user = $user;
        $this->meetup= $meetup;
    }

    public function getUser() {
        return $this->user;
    }

    public function getMeetup() {
        return $this->meetup;
    }
}
```

Jest to bardzo prosta klasa, której jedynym zadaniem jest przechowywanie danych o zdarzeniu dotyczącym spotkania i użytkownika. Teraz spowodujemy wyzwolenie tego zdarzenia, gdy użytkownik zapisze się na jakieś spotkanie. Wpisz poniższy kod w kontrolerze, za kodem sprawdzającym formularz:

```
if ($form->isValid()) {
    $meetup->addAttendee($user);
    // To jest nowy wiersz.
    $this->get('event_dispatcher')->dispatch(
        'meetup.join',
        new MeetupEvent($user, $meetup)
    );
    $em->flush();
}
```

Wystarczyło znaleźć usługę `event_dispatcher` i rozesłać zdarzenie `meetup.join` z porcją danych. Rozsyłanie zdarzenia to po prostu wysłanie wiadomości pod pewną nazwą, w tym przypadku `meetup.join`, z potencjalnymi danymi. Zanim kod przejdzie do wykonywania następnego wiersza, wszystkie klasy i obiekty nasłuchujące tego zdarzenia również mogą wykonać jakieś instrukcje.

> Nazwy zdarzeń dobrze jest przyporządkowywać do przestrzeni nazw, aby uniknąć ewentualnych kolizji. Zazwyczaj do oddzielania przestrzeni nazw zdarzeń używa się kropki i dlatego można spotkać zdarzenia w stylu `acme.user.authentication.success`, `acme.user.authentication.fail` itd.

Innym dobrym zwyczajem jest katalogowanie i dokumentowanie swoich zdarzeń. Z doświadczenia wiem, że jeśli dodaje się wiele zdarzeń, „bo tak łatwo się je wyzwala, gdyż to przecież

tylko nazwy", to po pewnym czasie trudno je wszystkie zapamiętać i łatwo się pogubić, do czego służą. Katalogowanie zdarzeń nabiera szczególnego znaczenia, gdy ktoś planuje udostępniać swój kod innym programistom. Wówczas należy utworzyć statyczną klasę zdarzeń:

```
namespace Khepin\BookBundle\Event;

final class MeetupEvents
{
    /**
     * Zdarzenie meetup.join jest wyzwalane, gdy użytkownik
     * rejestruje się na spotkaniu.
     *
     * Procedury nasłuchujące otrzymują egzemplarz obiektu:
     * Khepin\BookBundle\Event\MeetupEvent
     */
    const MEETUP_JOIN = 'meetup.join';
}
```

Jak napisałem, klasa ta służy jedynie do celów dokumentacyjnych. Kod w kontrolerze można zmienić następująco:

```
$container->get('event_dispatcher')->dispatch(
    MeetupEvents::MEETUP_JOIN,
    new MeetupEvent($user, $meetup)
);
```

Wiemy już, jak wyzwolić zdarzenie, ale jak na razie, nie mamy z tej wiedzy większego pożytku! Dodamy więc trochę więcej kodu. Najpierw utworzymy klasę nasłuchującą, która będzie odpowiedzialna za generowanie dla użytkownika nowej listy preferowanych spotkań:

```
namespace Khepin\BookBundle\Event\Listener;
use Khepin\BookBundle\Event\MeetupEvent;

class JoinMeetupListener
{
    public function generatePreferences(MeetupEvent $event) {
        $user = $event->getUser();
        $meetup = $event->getMeetup();
        // Kod generujący nowe preferencje użytkownika.
    }
}
```

Jest to zwykła klasa PHP. Nie musi ona niczego specjalnego rozszerzać, a więc nie musi też mieć jakiejś konkretnej nazwy. Najważniejsze, żeby zawierała jedną metodę przyjmującą argument `MeetupEvent`. Gdybyśmy teraz wykonali kod, nic by się nie stało, ponieważ jeszcze nie powiedzieliśmy, że ta klasa ma nasłuchiwać jakichkolwiek zdarzeń. W tym celu musimy zamienić ją w usługę. Oznacza to, że naszej procedurze nasłuchowej będzie można przekazać egzemplarz usługi geolokacyjnej, którą zdefiniowaliśmy w pierwszej części rozdziału, lub dowolnej innej usługi dostępnej w Symfony. Ponadto w definicji naszej procedury jako usługi zaobserwujemy też bardziej zaawansowane techniki użycia usług:

```
join_meetup_listener:
    class: Khepin\BookBundle\Event\Listener\JoinMeetupListener
    tags:
        - { name: kernel.event_listener, event: meetup.join,
         method: generatePreferences }
```

Sekcja `tags` oznacza, że przy pierwszym utworzeniu usługi `event_dispatcher` zostaną wyszukane i zapamiętane także inne usługi, którym przypisano określony znacznik (w tym przypadku `kernel.event_listener`). Jest to wykorzystywane również przez inne składniki Symfony, np. system formularzy (omówiony w rozdziale 3.).

Poprawianie wydajności

Osiągnęliśmy pewien cel przy użyciu zdarzeń i procedur nasłuchujących. Cała logika dotycząca obliczania preferencji użytkownika znajduje się w osobnej klasie nasłuchowej. Nie przedstawiłem szczegółowo implementacji tej logiki, ale wiadomo już, że najlepiej wynieść ją poza kontroler i przekształcić w niezależną usługę z możliwością wywoływania w procedurze nasłuchującej. Im więcej będziesz używać Symfony, tym bardziej oczywiste będzie Ci się to wydawać. Cały kod, który można przenieść do usługi, należy przenieść do usługi. Niektórzy programiści rdzenia Symfony twierdzą, że nawet kontrolery powinny być usługami. Jeśli zastosujesz się do tych wskazówek, Twój kod będzie łatwiejszy do testowania.

Kod działający po odpowiedzi

Gdy witryna stanie się bardziej skomplikowana i będzie miała dużo użytkowników, obliczenia preferowanych typów zdarzeń użytkowników mogą się dłużyć. Poza tym użytkownik może mieć przyjaciół na naszej stronie, w związku z czym chcielibyśmy, aby jego wybory miały wpływ także na preferencje jego znajomych.

W nowoczesnych aplikacjach sieciowych często nie trzeba czekać na zakończenie czasochłonnych operacji, zanim zostanie zwrócona odpowiedź do użytkownika. Oto niektóre z takich przypadków:

- Po wysłaniu filmu na serwer użytkownik nie powinien czekać na zakończenie konwersji tego filmu na inny format, aż pojawi się strona z informacją, że wysyłanie zakończyło się pomyślnie.
- Kilka sekund można zyskać, jeśli nie będzie się zmieniać rozmiaru obrazu profilowego użytkownika przed wyświetleniem informacji, że aktualizacja się powiodła.
- W naszym przypadku użytkownik nie powinien czekać na potwierdzenie, aż roześlemy wszystkim jego znajomym informację, że zapisał się na jakieś spotkanie.

Problemy te można rozwiązać na wiele sposobów, aby odciążyć proces generowania odpowiedzi. Można codziennie obliczać preferencje użytkownika za pomocą procesów wsadowych, ale to spowoduje opóźnienia w zwracaniu odpowiedzi, ponieważ aktualizacje będą wykonywane tylko raz dziennie, oraz może to prowadzić do marnowania zasobów. Można też użyć kolejki wiadomości i robotników w taki sposób, że kolejka powiadamiałaby robotników o konieczności

zrobienia czegoś. Byłoby to coś podobnego do rozwiązania ze zdarzeniami, ale kod wykonujący obliczenia działałby w innym procesie, a może nawet na innej maszynie. Nie trzeba by było też czekać na jego zakończenie, aby móc kontynuować.

W Symfony problem ten można łatwo rozwiązać, pozostając cały czas w systemie. Nasłuchując zdarzenia `kernel.terminate`, możemy uruchomić metodę naszej procedury nasłuchującej po tym, jak odpowiedź zostanie wysłana do klienta.

Zmienimy nasz kod, aby skorzystać z tej możliwości. Nasza nowa procedura nasłuchująca będzie teraz zachowywać się tak, jak napisano w poniższej tabeli:

Zdarzenie	Procedura nasłuchująca
`meetup.join`	Zapamiętuje użytkownika i spotkanie na później. Brak jakichkolwiek obliczeń.
`kernel.terminate`	Generuje preferencje użytkownika. Wykonuje obliczenia.

Nasz kod powinien teraz wyglądać tak:

```
class JoinMeetupListener
{
    protected $event;

    public function onUserJoinsMeetup(MeetupEvent $event) {
        $this->event = $event;
    }

    public function generatePreferences() {
        if ($this->event) {
            // Generuje nowe preferencje użytkownika.
        }
    }
}
```

Następnie musimy też zmienić konfigurację, aby wywoływała `generatePreferences` w przypadku wystąpienia zdarzenia `kernel.terminate`:

```
join_meetup_listener:
        class: Khepin\BookBundle\Event\Listener\JoinMeetupListener
        tags:
            - { name: kernel.event_listener, event: meetup.join,
              method: onUserJoinsMeetup }
            - { name: kernel.event_listener, event:
              kernel.terminate, method: generatePreferences }
```

Wystarczyło dodać znacznik do istniejącej procedury nasłuchowej. Jeśli rozważałeś utworzenie nowej usługi tej samej klasy, tylko nasłuchującej innego zdarzenia, teraz będziesz mieć dwa różne egzemplarze usługi. W związku z tym usługa, która zapamiętała zdarzenie, nigdy

nie zostanie wywołana w celu wygenerowania preferencji, a usługa wywołana w celu wygenerowania preferencji nigdy nie otrzyma zdarzenia do pracy. Dzięki tej nowej konfiguracji kod wykonujący intensywne obliczenia nie przeszkadza już w wysyłaniu odpowiedzi do użytkownika, który może cieszyć się komfortowym przeglądaniem stron.

Podsumowanie

W niniejszym rozdziale zostały wprowadzone dwa podstawowe pojęcia systemu Symfony, zwłaszcza jeśli chodzi o tworzenie rozszerzeń. Na przykładzie geokodowania dowiedziałeś się, jak łatwo dodaje się usługi podobne do standardowych usług systemu. Ponadto pokazałem, jak za pomocą zdarzeń odpowiednio rozdysponować logikę programu, aby nie zaśmiecić kontrolerów niechcianym kodem. Na zakończenie przy użyciu zdarzeń przyspieszyliśmy działanie witryny i uczyniliśmy przeglądanie stron bardziej komfortowym.

Możesz wierzyć lub nie, ale jeśli dobrze zrozumiesz działanie zdarzeń i usług, to będziesz wiedzieć prawie wszystko na temat rozszerzania Symfony. W dalszej części książki będziemy wielokrotnie wracać do tych dwóch pojęć, a więc jest bardzo ważne, aby je dobrze zrozumieć.

W następnym rozdziale dodamy nowe polecenia do narzędzia konsolowego Symfony oraz dostosujemy do swoich potrzeb silnik szablonów. W tym również bardzo pomocne będą usługi.

Skorowidz

A

abstrakcyjna definicja usługi, 72
ACL, access control list, 74
adnotacja, 74, 80
    @Annotation, 80
    @ORM\Version, 99
aktualizowanie
    preferencji użytkownika, 27
    wersji, 100
API, 85
aplikacja GitHub, 110
atak typu CSRF, 27, 87
atrapy klas, 19
atrybuty znacznika, 24
autoryzacja, 63, 74
awatar, 33

B

baza danych
    MongoDB, 89, 90
    MySQL, 89
    PostgreSQL, 89
bezpieczeństwo, 63
biblioteka Imagine, 34
błąd, 44

D

dane
    użytkowników, 56
    zdarzenia, 62
definiowanie
    adnotacji, 80
    usługi, 24
DIC, dependency injection container, 17
Doctrine, 25, 89
dodawanie
    adnotacji, 82
    mapy do widoku, 51
    pól, 60
dokumentacja, 116
dokumentowanie zdarzeń, 28
dostawca użytkowników, 68
dostęp do geokodera, 15
dystrybucja, 118
dziedziczenie usługi, 72

F

fabryka zabezpieczeń, 68, 108
filtr, 103
    różnicy czasowej, 44
formularz, 27, 47
    jako usługa, 56
funkcja
    buildView(), 56
    configure(), 34
    execute(), 34
    parse(), 96

G

geokoder, 21
geolokalizacja, 14

H

hasło, 67

I

inicjacja geokodowania, 15
integracja z mapami Google, 48
interfejs
  API, 85
  do usług, 38
  Geocoder, 21
  PHP, 104
  UserOwnedEntity, 104
  VoterInterface, 75

J

jednostka robocza, 101

K

klasa
  adaptacyjna, 16
  Address, 58
  adnotacyjna, 80
  AuthenticationListener, 68, 110
  AuthenticationProvider, 71
  BaseController, 17
  Coordinate, 48, 55
  dostawcza, 16
  Form, 57
  geokodowania, 16
  KhepinGitAuthBundle, 108
  OwnerFilter, 105
  PHPUnit_Framework_TestCase, 19, 37
  Token, 67
  Type, 51, 90
  UserLocator, 19
  UserProvider, 68, 73
  Voter, 75
  WebTestCase, 19, 37
kod działający po odpowiedzi, 30
kompilacja DIC, 23
konfiguracja
  formularza, 49
  ORM, 104
konsola, 36
kontener wstrzykiwania zależności, 17
kontrola wersji, 97

L

licencja MIT, 119
licencjonowanie, 118
lista
  kontroli dostępu, 74
  pakietów, 116
lokalizacja predefiniowana, 55

M

mapowanie, 92
  obiektowo-relacyjne, 89
mapy Google, 47
metoda
  attemptAuthentication, 66
  closureToDatabase, 90
  closureToPHP, 90, 91
  convertToDatabaseValue, 90
  convertToPHPValue, 90
  createView, 51
  get*Annotations, 82
  getClass, 82
  getForm, 51
  getKey, 70
  getUserCoordinate, 57
  prePersist, 99
  preUpdate, 99
  reverseTransform, 54
  supportClass, 75
  supportsAttribute, 75
  Trait, 98
  transform, 54
  vote, 75
modyfikowanie formularza, 60, 61

N

narzędzia
  do geokodowania, 14
  do testowania, 117
narzędzie
  do mapowania obiektowo-relacyjnego, 89
  Mockery, 117
nazwa przestrzeni nazw, 108
nazwy zdarzeń, 28

O

odczytywanie adnotacji, 82
ODM, object-document mapper, 89
ODM Mongo, 92
opcja compound, 50
ORM, 89

P

pakiet
  FOSUserBundle, 72
  ODM Mongo, 92
  SensioFrameworkExtraBundle, 82
parser, 95
plik
  bootstrap.php, 117
  config.yml, 19, 94, 109
  Configuration.php, 113
  Extension.php, 113
  phpunit.xml, 117
  README, 116
  routing.yml, 66
pliki zabezpieczeń, 72
polecenia, 33
  jako interfejs do usług, 38
polecenie picture:resize, 37
procedura nasłuchująca uwierzytelniania, 66
procedury nasłuchujące, 25, 62
przekształcanie danych, 54
przestrzeń nazw, 108

R

relacja wiele do wielu, 26
reprezentacje danych, 54
rozmiar obrazów, 34

S

sekcja tags, 30
serwer relacyjnych baz danych, 10
serwis GitHub, 64, 66
składnia funkcji DQL, 96
skrypt, 41
struktura pakietu, 108
system
  sqlite, 101
  Symfony, 10
  szablonów Twig, 40
szablony Twig, 40

T

technologia OAuth, 64
testowanie
  bazy danych, 101
  mapowania, 92
  pakietu, 116
  polecenia, 37
  rozszerzeń Twig, 43
  usług, 19
token, 68, 72
token DISTANCE, 95
transformatory danych, 54
Twig, 40, 41
  rozszerzenia, 40
  testowanie rozszerzeń, 43
tworzenie
  filtra Doctrine, 103
  formularzy, 47
  map, 51
  pakietu, 107, 113
  własnych typów danych, 89

U

udostępnianie
  konfiguracji, 110
  pakietu, 116
  własnych rozszerzeń, 107
uprawnienia szczegółowe, 75
usługa, service, 13
  event_dispatcher, 28
  fos_user.user_manager, 38
  geolokalizacji, 14, 29
  ivory_google_map.map, 51
  shrinker, 39
  user_locator, 24, 56, 57
usługi domyślne, 13
ustawianie wersji, 99
usuwanie pól, 60
uwierzytelnianie, 63
  OAuth poprzez GitHub, 64
używanie
  mapy, 51
  wersji, 100

V

Votery, 75

W

Walker AST, 103
warstwa abstrakcji baz danych, 97
wersjonowanie, 100
weryfikowanie poprawności pakietu, 113
widżet wyświetlający mapę, 47, 50
wiersz poleceń Composera, 33
własne
  funkcje DQL, 93
  funkcje SQL, 93
  rozszerzenia, 107
  typy danych, 89
  zdarzenia, 27
współrzędne geograficzne, 47
wstrzykiwanie zależności, 17
wydajność, 30
wyjątek, 51
wyświetlanie
  listy usług, 13
  formularza, 59
  pytań, 36
wyzwalanie zdarzeń, 26
wzorzec Obserwator, 25
wzór na odległość punktów, 93

Z

zabezpieczanie
  API, 85
  kontrolerów, 83
zakres
  container, 18
  prototype, 18, 51
  request, 18
zapora ogniowa, 65, 68
zarządzanie skryptami, 41
zdarzenia własne, 27
zdarzenie
  kernel.controller, 25
  kernel.exception, 25
  kernel.request, 25, 105
  kernel.response, 25
  kernel.terminate, 25, 31
  kernel.view, 25
  loadClassMetadata, 98
  meetup.join, 31
  onClear, 98
  post*, 98
  POST_SET_DATA, 57
  POST_SUBMIT, 57
  postFlush, 98
  postLoad, 98
  PRE_SET_DATA, 56
  PRE_SUBMIT, 57
  prePersist, 98
  preRemove, 98
  preUpdate, 98
  SUBMIT, 57
zmiany w tokenie, 71
zmienianie rozmiaru obrazów, 34
znacznik <script>, 42
znakowanie usług, 21

Symfony2 Rozbudowa frameworka

Symfony2 to jeden z najpopularniejszych szkieletów do tworzenia aplikacji internetowych w języku PHP. Programiści PHP docenili jego możliwości, łatwość konfiguracji oraz elastyczność i wybierają go do najbardziej zaawansowanych projektów. Wokół tego szkieletu stworzyli również silną społeczność, która aktywnie wspiera początkujących programistów. Chcesz się przekonać, jak wykorzystać Symfony2 w codziennej pracy? Zastanawiasz się, jak rozszerzyć możliwości tego systemu i jeszcze bardziej dostosować go do własnych potrzeb? Jeżeli tak, to to książka dla Ciebie!

Znajdziesz w niej cenne porady na temat tworzenia usług, szablonów oraz formularzy. Dowiesz się, jak zwiększyć bezpieczeństwo Twojej aplikacji za pomocą uwierzytelnienia OAuth oraz własnych adnotacji. Poznasz Doctrine oraz zobaczysz, jak udostępnić stworzone rozszerzenie innym programistom. W tej książce znajdziesz również informacje na temat automatycznego testowania stworzonego kodu oraz budowania dokumentacji. Jest to doskonała pozycja dla programistów chcących w pełni wykorzystać możliwości szkieletu Symfony2!

Dzięki tej książce:

- stworzysz przydatną usługę
- zabezpieczysz dostęp do przygotowanego API
- wykorzystasz OAuth do uwierzytelnienia użytkownika
- udostępnisz stworzone rozszerzenie innym programistom

Rozszerz potencjał szkieletu Symfony2!

[PACKT] open source PUBLISHING community experience distilled

Helion

28198 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-283-0294-5

Informatyka w najlepszym wydaniu

cena: 32,90 zł