helion.pl

Idź do
• Spis treści
• Przykładowy rozdział

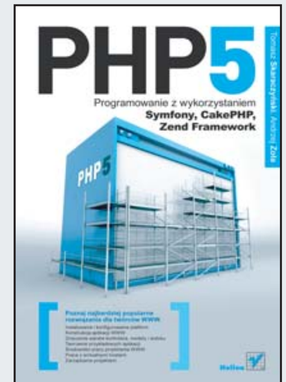
Katalog książek
• Katalog online
• Zamów drukowany katalog

Twój koszyk
• Dodaj do koszyka

Cennik i informacje
• Zamów informacje o nowościach
• Zamów cennik

Czytelnia
• Fragmenty książek online

PHP5. Programowanie z wykorzystaniem Symfony, CakePHP, Zend Framework



Autorzy: Tomasz Skaraczyński, Andrzej Zoła
ISBN: 978-83-246-2521-5
Format: 158×235, stron: 360

Na rynku dostępnych jest obecnie mnóstwo rozwiązań umożliwiających szybkie tworzenie serwisów internetowych z wykorzystaniem najpopularniejszego języka skryptowego, czyli PHP, oraz zestawów narzędzi składających się z bazy danych i serwera HTTP, takich jak MySQL i Apache. Wybór najlepszego oprogramowania dla konkretnej witryny może być czasami bardzo trudny, ponieważ każda z platform ma swoje wady i zalety. Sprawę może jednak ułatwić lektura odpowiedniej książki.

Książką tą jest „PHP5. Programowanie z wykorzystaniem Symfony, CakePHP, Zend Framework". Dokładnie opisano w niej sposób działania poszczególnych platform, zwracając szczególną uwagę na mocne i słabe strony każdego z rozwiązań. Na podstawie praktycznych przykładów zastosowań będziesz mógł samodzielnie przekonać się, które oprogramowanie najlepiej sprawdzi się w Twoim projekcie. Nauczysz się też wiele o budowie frameworków, poznasz znaczenie warstw i zasady administrowania serwisami, a ponadto zdobędziesz wiadomości na temat wirtualnych hostów i odpowiedniego środowiska pracy projektanta WWW.

- Instalowanie i konfigurowanie platform
- Konstrukcja aplikacji WWW
- Znaczenie warstw kontrolera, modelu i widoku
- Tworzenie przykładowych aplikacji
- środowisko pracy projektanta WWW
- Praca z wirtualnymi hostami
- Zarządzanie projektami

Poznaj najbardziej popularne rozwiązania dla twórców WWW

Kontakt
Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

Spis treści

Rozdział 1. Szybki start 9
Struktura serwisu 9
Ruszamy z projektem 10
Potrzebna aplikacja 11
Tworzymy moduł 12
Pierwsza akcja 13
Szablony akcji 15
Instalacja layoutu 16
Sprzątanie wewnątrz layoutu 17
Konfiguracja widoku aplikacji 17
Edycja pliku layoutu 18
Prezentowanie wyniku akcji 20
Brakujące elementy serwisu 21
Powiązanie akcji z menu 22
Stopka — i to by było na tyle 22
Podsumowanie 23

Rozdział 2. Warstwa kontrolera 25
Budowa aplikacji Symfony 25
Kontroler frontowy 26
Jak startuje aplikacja Symfony? 27
Czy można używać więcej niż jednego kontrolera frontowego? 28
Jak użyć innego kontrolera? 28
Co to jest środowisko pracy kontrolera? 28
Gdzie są konfigurowane środowiska pracy? 29
Czy można utworzyć własne środowisko pracy? 29
Akcje 29
Pliki z akcją 31
W jaki sposób przekazać parametr do akcji? 32
Zaglądamy do środka akcji 33
Przesyłanie parametrów w żądaniu HTTP 34
Czy formularze można tworzyć inaczej? 36
Szablon widoku 37
Co musisz wiedzieć na początek? 37
Jak sterować widokami? 37
Czy mogę używać własnych widoków? 38

Co z akcjami, które nie mogą być prezentowane w przeglądarce? ... 38
Warunkowe zwracanie widoków ... 39
Przekierowania ... 40
Żądanie nietrafione ... 42
Inne rodzaje przekierowań ... 44
Przed akcją i po akcji ... 44
Obiekt obsługujący żądania ... 46
Informacje o żądaniu ... 48
Informacje o zasobie ... 48
ParameterHolder i funkcje proxy ... 49
Funkcje proxy ... 50
Ciasteczka ... 51
Przesyłanie plików na serwer ... 52
Obsługa sesji ... 53
Proste logowanie ... 54
Usuwanie zmiennej z sesji ... 56
Zmienne sesji w widokach ... 57
Atrybuty jednorazowe ... 57
Kilka słów o konfiguracji sesji ... 59
System uprawnień ... 60
Przegląd funkcji systemu uprawnień ... 64
Zaawansowane listy uwierzytelnień ... 64
Walidacja ... 66
Mechanizm walidacji ... 67
Podsumowanie ... 68

Rozdział 3. Warstwa modelu ... 69

Od bazy do modelu ... 69
Baza danych ... 70
Generowanie schematu YML na podstawie bazy danych ... 73
Konfiguracja propela ... 73
Generowanie bazy danych na podstawie schematu YML ... 77
Anatomia pliku schema.yml ... 79
Dostępne typy danych ... 81
Definiowanie pól ... 83
Indeksy ... 84
Właściwości połączenia ... 84
Dwa schematy. Czy to możliwe? ... 85
Co w modelu piszczy ... 87
Katalogi modelu ... 88
Model w akcji ... 88
Konstruowanie kryteriów ... 100
Warunkowe pobieranie danych ... 100
Typy porównywania dozwolone dla metody add ... 102
Inne metody obiektu Criteria ... 103
Zliczanie rekordów ... 107
Surowe zapytania SQL ... 107
Korzystanie z Creole ... 108
Rozszerzanie modelu ... 109
Połączenia z bazą danych ... 112
Więcej o pliku database.yml ... 112
Podsumowanie ... 114

Rozdział 4. Warstwa widoku ... 115
Domyślna akcja i jej widok ... 115
Reguły dla szablonów widoku ... 116
Logika a szablon ... 117
Pomocniki ... 117
Pomocniki ogólnie dostępne ... 119
Layouty ... 120
Inny layout ... 121
Pomocniki w layoutach ... 123
Zmiana layoutu dla modułu ... 123
Zmiana layoutu dla szablonu widoku ... 124
Zmiana layoutu dla akcji ... 125
Usuwanie layoutu ... 126
Elementy widoku ... 127
Proste dołączanie pliku ... 127
Partiale ... 128
Komponenty ... 133
Sloty ... 136
Konfiguracja ... 139
Pliki view.yml ... 139
Kaskada plików konfiguracyjnych ... 140
Obiekt Response ... 141
Sterowanie sekcją meta poprzez obiekt odpowiedzi ... 142
Pliki zewnętrzne ... 143
Pliki CSS i JS ... 144
Manipulowanie kolejnością dołączanych plików ... 144
Określanie medium ... 145
Komponenty slotowe ... 146
Podsumowanie ... 149

Rozdział 5. Przykładowa aplikacja ... 151
Świat wizytówek ... 151
Projekt bazy danych ... 152
Instalacja layoutu i konfiguracja widoku ... 154
Wykonanie modelu ... 158
Budowa menu ... 158
Strona o firmie ... 160
Panel administracyjny — o firmie ... 161
Interfejs użytkownika — o firmie ... 164
Strona referencji ... 164
Panel administracyjny — referencje ... 165
Interfejs użytkownika — referencje ... 175
Strony z ofertą ... 176
Panel administracyjny — kategorie ... 177
Panel administracyjny — produkty ... 179
Panel administracyjny — kategorie — ciąg dalszy ... 185
Panel administracyjny — zdjęcia ... 188
Interfejs użytkownika — oferta ... 191
Sentencje — panel administracyjny i interfejs użytkownika ... 197
Licznik odwiedzin ... 202
Podsumowanie ... 203

Rozdział 6. Aplikacja Zend ... 205
Szybka instalacja ... 205
Test instalacji ... 206
Po instalacji ... 206
Pierwsza akcja na rozgrzewkę ... 207
Konfiguracja projektu ... 208
Layout ... 209
Interfejs klienta ... 210
    Strona o firmie ... 210
    Menu ... 214
    Referencje ... 216
Oferta ... 217
    Submenu ... 220
    Kategoria ... 221
    Szczegóły produktu ... 223
    Dodatki ... 224
Panel administracyjny ... 229
    Inny layout dla panelu ... 229
    Zarządzanie stroną o firmie ... 230
    Administracja referencjami ... 234
    Kategorie ... 243
    Zarządzanie produktami ... 252
    Sentencje ... 264
Podsumowanie ... 269

Rozdział 7. Aplikacja CakePHP ... 271
Instalacja frameworka ... 271
Konfiguracja bazy danych ... 272
O firmie ... 273
    Model ... 273
    Kontroler ... 274
    Widok ... 275
Layout ... 275
Logowanie na ekranie ... 276
Menu ... 276
    Komponent ... 277
    Helper ... 277
Referencje ... 280
    Model referencji ... 280
Oferta ... 282
    Model na rozgrzewkę ... 282
    Oferta w poszczególnych kategoriach ... 285
    Szczegóły wizytówki ... 287
Sentencje ... 288
    Komponent ... 288
    Uruchomienie komponentu Sentencje ... 289
Licznik ... 290
    Komponent licznika ... 290
    Uruchamianie licznika ... 291
Panel administracyjny ... 292
    Zmiana layoutu ... 292
    Strona administracyjna o firmie ... 293
    Referencje ... 296
    Kategorie ... 304

Produkty ... 310
Dodawanie nowego produktu ... 312
Sentencje ... 324
Podsumowanie ... 327

Podsumowanie ... 329

Dodatek A Środowisko pracy web developera ... 331
Serwer HTTP ... 331
Interpreter PHP ... 332
Serwer baz danych ... 332
Wszystko w jednym, czyli scyzoryk ... 332
Środowisko projektowania baz danych ... 333
Edytory kodu ... 333
Przeglądarki ... 334
Narzędzia do pracy w grupie ... 334

Dodatek B Wirtualne hosty ... 337
Importowanie wirtualnych hostów do pliku konfiguracyjnego Apache ... 337
Definiowanie wirtualnych hostów ... 337
Wirtualny host dla lokalnego hosta ... 338
Konfiguracja systemu Windows ... 338

Dodatek C Szybka instalacja ... 341
Odtworzenie bazy danych ... 341
Zainstalowanie projektu Symfony, Zend i CakePHP ... 341

Dodatek D Zarządzanie projektem ... 343

Bibliografia ... 345

Skorowidz ... 347

Rozdział 4.
Warstwa widoku

W poprzednich rozdziałach mogłeś przeczytać, że Symfony opiera się na regułach MVC (*Model-View-Controller*). Warstwy modelu oraz kontrolera zostały tam szczegółowo omówione. W tym rozdziale dowiesz się wszystkiego, co jest niezbędne do poprawnego konstruowania widoków.

Zanim przejdziesz dalej, warto w tym miejscu przypomnieć, czym jest widok i jaki ma związek z poprzednio poznanymi warstwami. Otóż widok to reprezentacja wizualna wykonanych w akcji operacji. Jeżeli spojrzysz na widok w kontekście poprzednio poznanych warstw, to łatwo zauważysz, że:

- ♦ kontroler steruje wszystkimi operacjami (poprzez akcje),
- ♦ model implementuje logikę biznesową aplikacji (przetwarza dane),
- ♦ widok opakowuje otrzymane dane w kod HTML tak, aby powstała wizualizacja wykonanych operacji.

Dalsze ćwiczenia wykonywane będą w aplikacji *widok*. Musisz więc ją utworzyć. Potraktuj to jako ćwiczenie przypominające poznane poprzednio zagadnienia. Utwórz także moduł `szablony`, którego będziesz używał w pierwszych ćwiczeniach.

Domyślna akcja i jej widok

Akcja `index` jest domyślną akcją każdego modułu. Zauważ, że Symfony dodaje ją automatycznie do tworzonego modułu. Zawiera ona przekierowanie do akcji wyświetlającej informacyjny komunikat o utworzeniu modułu. Zmiana widoku na własny wymaga usunięcia linijki kodu:

```
$this->forward('default', 'module');
```

Jeżeli dodałeś już komentarz, możesz przystąpić do utworzenia domyślnego widoku dla modułu. Utwórz więc plik *indexSuccess.php* i dodaj do niego przykładowy kod. Może on wyglądać jak na listingu:

```
<h1>Witaj Swiecie</h1>
<p>Lorem ipsum dolor sit amet consectetuer fames...</p>
<p>Lorem ipsum dolor sit amet consectetuer fames...
Lorem ipsum dolor sit amet consectetuer fames...
Lorem ipsum dolor sit amet consectetuer fames...</p>
<p>Lorem ipsum dolor sit amet consectetuer fames...
Lorem ipsum dolor sit amet consectetuer fames...</p>
<p>Lorem ipsum dolor sit amet consectetuer fames...</p>
```

Udało Ci się utworzyć widok, który prezentowany będzie zawsze, kiedy użytkownik nie określi akcji, jaką należy wykonać. W prawdziwej aplikacji warto zadbać, żeby ten widok był czymś w rodzaju strony startowej dla modułu.

Reguły dla szablonów widoku

Tworząc szablony zgodnie z ideą zawartą w technologii MVC, należy pamiętać o przestrzeganiu kilu reguł. Najważniejsze z nich to:

- dopuszczalne jest używanie w szablonach instrukcji sterujących: `if`, `switch` itp.
- dopuszczalne jest używanie w szablonach instrukcji pętli: `for`, `while`, `foreach` itp.
- niedopuszczalne jest używanie w szablonach kodu przetwarzającego dane,
- szablony służą jedynie do wizualizacji wcześniej wykonanych w akcjach operacji.

Kilka słów wyjaśniających pozwoli Ci lepiej zrozumieć przedstawione powyżej aspekty. Szablony widoków z natury powinny być jedynie reprezentacją wizualną przetworzonych danych. Z tego powodu dopuszcza się w nich jedynie używanie prostych konstrukcji językowych. Dla przykładu, możesz sprawdzić, czy jakiś warunek jest prawdziwy, i w zależności od wyniku przedstawić pewną część widoku. Ponadto dla danych reprezentowanych przez tablice możesz bez obawy używać pętli. Konstrukcje te służą jedynie do sterowania przepływem danych, a nie do generowania danych. Pamiętaj natomiast, że używanie instrukcji PHP do oprogramowania jakiejkolwiek logiki jest w szablonach niepożądane. Jeżeli musisz coś wyliczyć, przetworzyć, zrób to w akcji!

Dlaczego nie możesz używać w widoku funkcji wbudowanych w PHP? Najprostszą odpowiedzią byłoby napisanie: bo tak każe **MVC**. Ponieważ jednak taka odpowiedź nie satysfakcjonowałaby mnie ani trochę, spróbuję w paru zdaniach wyjaśnić Ci, skąd wziął się w ogóle taki pomysł. Wyobraź sobie, że pracujesz w zespole nad aplikacją komercyjną. Aplikacja jest dosyć złożona, a Ty jesteś jednym z programistów. W zespole jest również grafik odpowiedzialny za projektowanie layoutów i przygotowanie kodu HTML dla akcji. Jeżeli dobrze określicie granice swoich wpływów, możecie ułatwić sobie pracę. Brak instrukcji PHP w kodach widoku spowoduje, że dla grafika czytanie kodu będzie zdecydowanie łatwiejsze. Zapoznasz go z kilkoma zagadnieniami, takimi jak: pętle, instrukcje if oraz składowe widoku Symfony, i będzie on mógł pracować nad widokami sam. Z drugiej strony jest to również olbrzymia zaleta dla Ciebie. Możesz skupić się na programowaniu akcji, na rozwiązywaniu zawiłości aplikacji, nie martwiąc się przy tym stylizacją wizualną tego, co robisz. Mam nadzieję, że takie wyjaśnienie w zupełności Ci wystarczy.

Logika a szablon

W każdym systemie implementującym MVC logika oddzielona jest od szablonu widoku. Separacja tych elementów aplikacji pozwala na większy komfort pracy dla zespołu składającego się z kilku pracowników. Ponadto zwiększa możliwości systemu oraz efektywność pracy, ponieważ pozwala na wielokrotne wykorzystywanie tego samego kodu. Lepiej to zrozumiesz na konkretnym przykładzie.

Wyobraź sobie, że programujesz pewną aplikację. W aplikacji wymagana jest klasa `Klient`, reprezentująca klientów firmy. Z klasą powiązany jest obiekt odpowiadający za komunikację z bazą danych. Ta część aplikacji realizuje logikę klienta. Zwróć uwagę na to, że logika zawsze jest taka sama. Z drugiej strony reprezentacja wizualna *Klienta* może być różna w różnych częściach serwisu. Wystarczy, że jako przykład rozważysz widok w profilu *Klienta* oraz widok w panelu *Administratora*. Zwykle różnią się one w znacznym stopniu. Wniosek płynący z tego jest oczywisty, możemy z klasy `Klient` korzystać wielokrotnie. Jedyne, co trzeba zrobić, to dopisać odpowiedni szablon widoku. Mam nadzieję, że coraz bardziej dostrzegasz zalety takiego programowania.

Pomocniki

Pomocniki Symfony są funkcjami napisanymi w języku PHP. Realizują bardzo standardowe zadania, przez co ułatwiają konstruowanie widoków. Dla przykładu, używając odpowiedniego pomocnika Symfony, możesz wstawić na stronę edytor **WYSIWYG** (*What You See Is What You Get* — to, co widzisz, jest tym, co otrzymasz), który pozwoli Ci łatwo formatować teksty. Inną ważną cechą pomocników jest to, że wiele z nich jest zintegrowanych z mechanizmami Symfony. Dzięki temu tworzenie hiperłączy do akcji jest bardzo łatwe. Wystarczy, że wywołasz odpowiedni pomocnik, podasz odpowiednie parametry, a dostaniesz na wyjściu kod uwzględniający ustawienia frameworka (np. mechanizmy przepisywania linków).

Wszystkie pomocniki zajmujące się daną tematyką umieszczane są w jednym pliku. Nazwa każdego pliku kończy się tekstem *Helper.php*. Ze względu na to bardzo często, mówiąc o pomocniku, ma się na myśli całą grupę funkcji. Warto w tym miejscu zastanowić się, gdzie Symfony przechowuje pliki pomocników? Nie musisz głęboko szukać. Wystarczy, że będąc w katalogu projektu, wejdziesz do *lib/symfony/helper*. W tym miejscu znajduje się odpowiedź na postawione pytanie.

Co prawda nie zrobiłeś zbyt wielu ćwiczeń w module `szablony`, warto jednak w tym miejscu zrobić nowy moduł. Utwórz więc moduł `pomocniki`, co pozwoli Ci w przyszłości łatwiej odnajdywać potrzebne elementy. Na początek przypomnimy sobie kilka istotnych rzeczy. W tym celu dodaj do modułu akcję `linkDoZasobu`. Akcja nie zawiera żadnego kodu. Poniżej znajduje się listing jej widoku:

```
<div>
  <h1>Wynik pomocnika url_for</h1>
  <?php echo url_for('/pomocniki/linkDoZasobu?pokaz=jablko'); ?><br/>
  <?php echo url_for('/pomocniki/linkDoZasobu?pokaz=gruszka', true); ?><br/>
```

```
</div>
<hr/>
<div>
```

W przykładzie użyłeś pomocnika `url_for`. Jego działanie jest bardzo proste. Jako argument podajesz tekst wskazujący zasób, w wyniku zaś dostajesz adres odpowiedni dla Symfony. Zauważ, że format tego tekstu musi być taki jak poniżej:

```
moduł/akcja?parametr1=wartość1&parametr2=wartość2...
```

Tekst zostanie przekształcony tak, żeby uwzględnić Twoją konfigurację Symfony.

Spróbuj teraz napisać inny przykład. Zacznij od utworzenia akcji `dolaczPomocnik`. Poniżej jest jej widok:

```
<div>
  <h3>Wynik działania auto_link_text</h3>
  <?php
    use_helper('Text');
    echo auto_link_text(
      'Aby umieścić ofertę w serwisie http://www.allegro.pl
         należy założyć konto użytkownika na stronie
         http://allegro.pl/new_user.php'
    );
  ?>
<div>
```

Co robi pomocnik `auto_link_text`? Wyszukuje w tekście poprawne adresy URL i zamienia je na hiperłącza. Działanie pomocnika jest więc dosyć proste. O wiele ciekawsza jest instrukcja `use_helper`. Przyjmuje ona jako parametr nazwę grupy pomocników, którą należy dołączyć do widoku. Dlaczego więc poprzednio nigdy jej nie wywoływałeś? Pewne grupy najpowszechniej stosowanych pomocników dołączane są do widoków domyślnie. Jeżeli potrzebujesz mniej popularnych, musisz je dołączyć za pomocą `use_helper`. Zwróć uwagę na to, że wystarczy podać nazwę grupy. Nie musisz podawać całej nazwy pliku. Jeżeli parametrem jest wartość `Text`, do szablonu zostanie dołączony plik *TextHelper.php*.

Ćwiczenie 4.1

Utwórz pomocnik *FileHelper.php*. Pomocnik zawiera jedną funkcję o nazwie **storage**. Przyjmuje ona jako parametr wielkość będącą rozmiarem pliku w bajtach. Funkcja zwraca jako wynik rozmiar pliku w najlepiej dopasowanych jednostkach.

Rozwiązanie

W tym ćwiczeniu nauczysz się, w jaki sposób tworzyć własne pomocniki. Na początek w katalogu modułu `pomocniki` otwórz katalog *lib*. Następnie utwórz w nim folder o nazwie *helper*. W kolejnym kroku musisz utworzyć plik *FileHelper.php*. Jeżeli udało Ci się wszystko wykonać poprawnie, otwórz plik helpera do edycji. Musisz teraz dopisać do niego funkcję realizującą założenia z ćwiczenia. Poniżej przykładowa implementacja:

```
<?php
  function storage( $size ) {
    $aUnits = array('B', 'KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB');
    $unitIndex = 0;
    while ($size >= 1024) {
      $size /= 1024;
      $unitIndex++;
    }
    return round( $size, 1 ).' '.$aUnits[$unitIndex];
  }
?>
```

Spróbuję teraz w paru słowach wyjaśnić, co robi funkcja. Po pierwsze, ustala indeks odpowiedniej jednostki. Po drugie, zmniejsza obecną wielkość tak, żeby pasowała do ustalonej jednostki. Analizując algorytm, pamiętaj, że 1 KB = 1024 B, 1 MB = 1024 KB itd.

Teraz nadszedł czas na wypróbowanie utworzonego właśnie helpera W tym celu musisz utworzyć akcję `wlasnyPomocnik`. Widok akcji przedstawiony jest na listingu:

```
<?php
  use_helper('File');
  echo storage(1024),'<br />';
  echo storage(1024*1024),'<br />';
  echo storage(1024*1024*1024),'<br />';
?>
```

Jak widzisz, własne pomocniki dodaje się tak samo jak pozostałe (niedomyślne). Jeżeli pomocnik zadziała dobrze, to na ekranie powinieneś zobaczyć następujący wynik:

```
1 KB
1 MB
1 GB
```

Pamiętaj, żeby tworzyć pomocniki wtedy, kiedy są naprawdę potrzebne, tj. zawierają funkcje, których używasz wielokrotnie w różnych częściach aplikacji.

Pomocniki ogólnie dostępne

W poprzedniej sekcji mogłeś poczytać o tym, że część pomocników dołączana jest do widoku domyślnie. Poniżej znajdziesz ich listę wraz z krótkim wyjaśnieniem:

- `Helper` — pomocnik odpowiadający za dołączanie innych pomocników,
- `Tag` — pomocniki bazowe dla innych pomocników (zawierają funkcje tworzące znaczniki HTML),
- `Url` — pomocniki konstruujące adresy URL,
- `Asset` — pomocniki tworzące ścieżki do zasobów oraz znaczniki wymagające ścieżek do zasobów (`<img>`),
- `Partial` — pomocniki pozwalające podzielić widok na fragmenty zwane partialami (dowiesz się o nich więcej w dalszej części),

- `Cache` — pomocniki zarządzające pamięcią podręczną,
- `Form` — pomocniki wspomagające tworzenie kontrolek formularzy HTML.

Łatwo zauważyć, że domyślna grupa pomocników oferuje całkiem spore możliwości. Ponadto zawiera elementy, które są zdecydowanie najczęściej używane przez programistów. Pamiętaj, jeżeli nie jesteś pewny, co robi dany pomocnik, warto zajrzeć do jego kodu. Zawiera on wiele wskazówek, które możesz wykorzystać podczas tworzenia własnych pomocników.

Layouty

Z punktu widzenia poprawności HTML szablony nie są kompletnym kodem. Nie zawierają całej struktury dokumentu, a jedynie pewną jego cześć. Z drugiej strony szablony pozwalają zachować niezależność od logiki oraz podzielić zadania na mniejsze części.

Ze względu na cząstkowość szablonów docelowo należy je opakować w strukturę dokumentu HTML. W ten sposób będą stanowiły razem kompletną stronę WWW. Symfony realizuje to zadanie za pomocą pliku zwanego **layoutem**. Zawiera on ogólną strukturę serwisu, ale nie zawiera jego treści. Treść tworzona jest przez szablony widoków i umieszczana w layoucie w przeznaczonych do tego celu miejscach. Więcej szczegółów poznasz, czytając dalszą część rozdziału.

Zanim przejdziesz dalej, powinieneś utworzyć nowy moduł o nazwie `layout` (pamiętaj o zakomentowaniu przekierowania akcji `index`). W module tym wykonasz wiele ćwiczeń, które zwiększą Twoją wiedzę i pozwolą lepiej operować elementami warstwy widoku. Zaczniemy od instalacji layoutu.

Ćwiczenie 4.2

W pliku *finanse_layout.zip* umieszczony jest pewien `layout`. Zainstaluj go jako `layout` domyślny dla całej aplikacji widok.

Rozwiązanie

Na początek rozpakuj archiwum *zip*. Przejdź do katalogu głównego Twojego projektu Symfony. Znajdziesz w nim katalog o nazwie *web*. Folder ten zawiera trzy istotne dla tego zadania podfoldery: *css*, *images* oraz *js*. W ich miejsce musisz przekopiować foldery (o takich samych nazwach) wypakowane wcześniej z archiwum.

Drugi krok instalacji wymaga powrotu do katalogu głównego projektu. Następnie przejdź do katalogu *apps/widok*. Znajdziesz tutaj folder o nazwie *templates*. Jest to miejsce, w którym Symfony przechowuje `layouty` Twojej aplikacji. Nie pozostaje Ci nic innego, jak przekopiować do *templates* layout wypakowany z archiwum *finanse_layout.zip*.

Ostatni krok polega na edycji pliku *view.yml* zawierającego konfigurację widoku dla aplikacji *widok*. W celu realizacji tego zadania otwórz plik do edycji. Znajdziesz go w *<hatalog_projektu>/apps/widok/config*. Plik zawiera sekcję

```
stylesheets:    [main]
```

Zmień tekst `main` na `style`. Musisz tak zrobić, ponieważ Twoje style zapisane są w pliku *style.css*, a nie w domyślnym pliku *main.css*. Teraz musisz przetestować instalację. W tym celu wpisz w przeglądarce

```
http://127.0.0.1/widok.php/layout
```

Jeżeli wszystkie polecenia wykonałeś poprawnie, na ekranie zobaczysz zawartość przedstawioną na rysunku 4.1.

Rysunek 4.1. *Widok instalacji szablonu finanse_widok*

Pamiętaj, domyślnie szablon używany jest dla wszystkich widoków akcji. Możesz to oczywiście zmienić, o czym przekonasz się w dalszej części książki.

Inny layout

Zmiana layoutu jest bardzo praktyczną rzeczą. Wiele serwisów praktykuje zmiany okolicznościowe. Dla przykładu, jeżeli mamy sklep internetowy, być może zechcemy mu nadać inny wygląd podczas świąt.

Ćwiczenie 4.3

Zainstalować layout `wizytowki` dla wszystkich modułów w aplikacji widok.

Rozwiązanie

Na początek musisz rozpakować archiwum *wizytowki.zip*. Zawartość archiwum jest następująca:

1) plik *wizytowki.php* — nowy layout,
2) plik *css/wizytowki.css* — style określające strukturę oraz wygląd elementów serwisu,
3) pliki *images/wizytowki/** — pliki graficzne używane przez style oraz layout.

Zwróć uwagę na to, że pliki dla layoutu wizytowki nie są umieszczone w katalogu *images*. Powód jest prosty. W ten sposób łatwo oddzielić pliki należące do poszczególnych layoutów.

Kolejny krok polega na przekopiowaniu pliku w odpowiednie miejsce struktury katalogów Symfony. Są to operacje identyczne w stosunku do omawianego poprzednio procesu instalacji. Musisz więc przekopiować:

1) plik *wizytówki.php* do katalogu *apps/widok/templates*,

2) plik *css/wizytowki.css* do katalogu *web/css*,

3) cały katalog *images/wizytowki* do katalogu *web/images*.

W ten oto sposób udało Ci się zainstalować layout. Teraz musisz poinformować Symfony o tym, który layout ma zostać użyty dla Twoich akcji. W tym celu otwórz do edycji plik *view.yml*. Znajdziesz go w katalogu *conf* umieszczonym w aplikacji *widok*. Poniżej znajdziesz listing pliku konfiguracyjnego:

```
default:
  http_metas:
    content-type: text/html

  metas:
    title:        Superwizytówki na każdą kieszeń.
    robots:       index, follow
    description:  Tanie wizytówki dla firm.
    keywords:     wizytówki, tanie wizytówki
    language:     pl

  stylesheets:    [wizytowki]

  javascripts:    []

  has_layout:     on
  layout:         wizytowki
```

Najważniejsze (z punktu widzenia instalacji layoutu) są dyrektywy `stylesheets` i `layout`. Pierwsza określa nazwę pliku ze stylami, jakiego należy użyć, druga zaś nazwę pliku layoutu, który należy zastosować. Pamiętaj, że po zmianie pliku *yml* należy usunąć pamięć podręczną. Możesz to zrobić przykładowo tak:

```
>symfony.bat cc
```

Argument `cc` jest skrótem dla `clear cache` (czyść pamięć podręczną).

Testowanie instalacji wymaga wywołania w przeglądarce adresu:

http://127.0.0.1/widok.php/layout.

Jeżeli wszystko przebiegło poprawnie, w oknie przeglądarki zobaczysz widok zaprezentowany na rysunku 4.2.

Rysunek 4.2. *Nowy layout*

Co, jeżeli „skończą się święta” i zechcesz wrócić do starego layoutu? Musisz zmienić konfigurację w pliku *view.yml*, wyczyścić pamięć podręczną i... to już wszystko.

Pomocniki w layoutach

Możesz używać w layoutach dowolnego zestawu pomocników. Na pierwszy ogień idą jednak zwykle te, które odpowiadają za generowanie znaczników dla sekcji HEAD. Otwórz na chwilę plik *wizytowki.php*. Na początku pliku znajdziesz instrukcje:

```
<?php
  include_http_metas();
  include_metas();
  include_title();
?>
```

Wszystkie użyte powyżej funkcje to pomocniki. Odpowiadają za umieszczenie w sekcji HEAD wartości podanych w pliku konfiguracyjnym *view.yml*. Wartościami tymi są nazwy plików CSS, nazwy plików JS, tekst tytułu itp. Zaletą takiego podejścia jest możliwość dostosowywania sekcji HEAD dla modułu a nawet widoku akcji. Przekonasz się o tym w dalszej części książki.

Zmiana layoutu dla modułu

Zmiana layoutu na poziomie modułu jest bardzo ważną cechą Symfony. W ten sposób możesz budować moduły jako niezależne części serwisu posiadające indywidualny widok (dostosowany do tematyki). Dla przykładu, w serwisach informacyjnych inaczej można przedstawiać dane o sporcie, inaczej o gospodarce itp.

Ćwiczenie 4.4

Utwórz moduł o nazwie `layout2` (w aplikacji widok). Określ *layout.php* jako domyślny dla całego modułu.

Rozwiązanie

Na początek utwórz moduł:

```
>symfony init-mod widok layout2
```

Przejdź do modułu, otwórz plik *action.class.php* i zakomentuj instrukcję umieszczoną w akcji `index`. Następnie w katalogu *layout2/config* utwórz plik o nazwie *view.yml*. Do pliku wprowadź konfigurację przedstawioną na listingu poniżej:

```
all:
  layout: layout
  stylesheets: [style]
```

Ponieważ zmiana dotyczy pliku YML, musisz wyczyścić pamięć podręczną. Na koniec wywołaj w przeglądarce poniższe adresy:

http://127.0.0.1/widok.php/layout

http://127.0.0.1/widok.php/layout2

Jeżeli wszystkie operacje wykonałeś poprawnie, w przeglądarce zobaczysz dwa różne layouty. I o to właśnie chodziło!

Zmiana layoutu dla szablonu widoku

Projektanci Symfony nie zatrzymali się na zmianie szablonu do modułu. Poszli zdecydowanie dalej, pozwalając programiście na wymianę layoutu dla konkretnego widoku.

Ćwiczenie 4.5

Utworzyć akcję `historia` w module `layout2`. Dla akcji ustawić layout `wizytowki`.

Rozwiązanie

W pliku *actions.class.php* utwórz pustą akcję historia. Ponadto w katalogu *templates* (modułu `layout2`) umieść pusty szablon widoku (jest to równoznaczne z utworzeniem pustego pliku *historiaSuccess.php*). Następnie dodaj do pliku *view.yml* poniższą konfigurację:

```
historiaSuccess:
  layout: wizytowki
  stylesheets: [wizytowki]
```

Pozostało jeszcze wyczyścić pamięć podręczną i wywołać żądanie akcji w przeglądarce.

No i jak wynik? Niestety, niezbyt miły dla oka. Co prawda akcja korzysta z właściwego layoutu, ale coś jednak nie gra. Pytanie tylko, co? Odpowiedź jest dosyć prosta. Ponieważ dla modułu domyślnym layoutem jest *layout.php*, wymieszały nam się style. Jak temu zapobiec? Właściwa konfiguracja na tym poziomie powinna wyglądać tak:

```
all:
  layout: layout
  stylesheets: [-wizytowki, style]
historiaSuccess:
  layout: wizytowki
  stylesheets: [-style, wizytowki]
```

Jak widzisz, jest ona prawie taka sama jak poprzednia. Jednak, jak powszechnie wiadomo, „prawie" robi wielką różnicę. Zwróć uwagę na minus przed niektórymi nazwami plików. Oznacza on, że w danym miejscu nie należy pliku dołączać do kodu wynikowego. Wyczyść więc pamięć podręczną, odśwież stronę w przeglądarce i voilà!

Zmiana layoutu dla akcji

W poprzedniej sekcji dowiedziałeś się, jak zmienić layout dla konkretnego widoku. Możesz w ten sposób ustawić inny layout dla szablonu sukcesu, inny dla szablonu błędu itp. Co jednak w przypadku, kiedy chciałbyś, aby szablony widoków jednej akcji korzystały z tego samego layoutu? Możesz oczywiście każdy widok konfigurować oddzielnie. Nie jest to nawet aż tak kłopotliwe. Istnieje jednak znacznie prostsza metoda, która pozwala przypisywać layout do wszystkich widoków akcji jednocześnie.

Ćwiczenie 4.6

Utwórz akcje `layoutDlaAkcji`. Każdy widok utworzonej akcji musi korzystać z layoutu o nazwie `wizytowki`.

Rozwiązanie

Utwórz pustą akcje `layoutDlaAkcji` oraz pusty szablon widoku. Wpisz w pasku adresu

http://127.0.0.1/widok.php/layout2/layoutDlaAkcji

i upewnij się, że akcja korzysta z pliku *layout.php*.

Kolejnym krokiem jest edycja akcji. Musisz do niej dopisać poniższą instrukcję:

```
$this->setLayout('wizytowki');
```

Metoda `setLayout` pozwala określić nazwę layoutu, jaki należy użyć dla akcji. Możesz teraz odświeżyć stronę w przeglądarce. Pewne zmiany już widać, jednak łatwo zauważysz, że coś jeszcze wymaga dopracowania. Łatwo się domyślić, że brakuje poprawnych stylów. Dopisz do akcji instrukcję:

```
$this->getResponse()->addStylesheet('wizytowki');
```

i ponownie odśwież stronę w przeglądarce. Efekt jest nieco lepszy, jednak nie do końca dobry. Metoda `addStylesheet` (wywoływana dla obiektu odpowiedzi — `sfWebResponse`)

dołącza plik ze stylami. Ponadto w układzie, który uzyskałeś, do widoku dołączany jest również plik *style.css* (konfiguracja dla całego modułu). Niestety, pliki te nieco sobie przeszkadzają. W Symfony 1.0 nie ma łatwego sposobu na usunięcie zbędnego pliku z odpowiedzi. Najprostszą metodą jest stworzenie odpowiedniego wpisu w pliku *view.yml*. W tym celu musisz dopisać poniższe dyrektywy:

```
layoutDlaAkcjiSuccess:
  stylesheets: [-style]
```

Teraz musisz jeszcze wyczyścić pamięć podręczną, odświeżyć stronę — i wszystko powinno być już na swoim miejscu. Co prawda było trochę problemów, jednak efekt jest zgodny z oczekiwanym.

Usuwanie layoutu

Niektóre akcje, jak na przykład żądania AJAX, nie wymagają layoutu. Powinieneś więc wiedzieć, w jaki sposób można go wyłączyć, jeżeli zajdzie taka potrzeba.

Ćwiczenie 4.7

Utwórz akcję `bezLayoutuYML`. Zablokuj dla niej dekorowanie layoutem.

Rozwiązanie

W pliku *actions.class.php* utwórz akcję `bezLayoutuYML`. Widok `bezLayoutuYMLSuccess` przedstawiony jest poniżej:

```
<div>Ta akcja nie potrzebuje dekorowania layoutem</div>
```

Następnie otwórz plik view.yml do edycji i dopisz do niego poniższe dyrektywy:

```
bezLayoutuYMLSuccess:
  layout: no
```

Wyczyść pamięć podręczną, a następnie odśwież stronę. Jeżeli wszystko zrobiłeś dobrze, w oknie przeglądarki zobaczysz jedynie napis:

```
Ta akcja nie potrzebuje dekorowania layoutem
```

Na koniec zajrzyj do kodu źródłowego. Łatwo zauważysz, że poza kodem, który wpisałeś w widoku akcji, niczego więcej w kodzie źródłowym nie ma.

Ćwiczenie 4.8

Utwórz akcję `brakLayoutuAkcja`. Zablokuj dla niej dekorowanie layoutem.

Rozwiązanie

Ćwiczenie, które teraz wykonasz, jest bardzo podobne do poprzedniego. Różnica będzie polegała na tym, że tym razem usuniemy layout z poziomu akcji. Na początek utwórz pustą akcję oraz widok sukcesu. W widoku wprowadź taką treść jak poprzednio. Następnie dopisz do akcji poniższą instrukcję:

```
$this->setLayout(false);
```

Podanie wartości `false` dla metody `setLayout` oznacza wyłączenie dekorowania layoutem. I o to właśnie nam chodziło.

Elementy widoku

Każdy element widoku ma dostęp do pewnego zbioru zmiennych Symfony. Do najważniejszych z nich należą:

- `$sf_context` — obiekt udostępniający kontekst żądania,
- `$sf_request` — obiekt udostępniający informacje o żądaniu,
- `$sf_params` — obiekt udostępniający parametry przesłane w żądaniu,
- `$sf_user` — obiekt udostępniający dane użytkownika (w tym sesje).

Wszystkie obiekty przedstawione powyżej poznałeś już wcześniej. Były one szeroko omówione w rozdziale 2. „Warstwa kontrolera". W dalszych częściach tego rozdziału nauczysz się, jak korzystać z tych obiektów w widoku.

Proste dołączanie pliku

W każdym serwisie WWW pewne elementy strony są wspólne dla różnych stron. Do takich elementów można zaliczyć stopkę, menu itp. Ponieważ dobrze zaprogramowany serwis nie powinien dublować kodu, zwykle elementy takie umieszczane są w osobnych plikach. Następnie pliki te dołączane są we właściwych częściach serwisu. Na początek zapoznasz się z prostym dołączaniem plików. W tym celu utwórz moduł `elementy_widoku`, a następnie rozwiąż poniższe ćwiczenie.

Ćwiczenie 4.9

Utwórz kopię layoutu *wizytowki.php*. Nowy plik nazwać *wizytowki_dolaczenie.php*. Następnie utwórz plik *menu.php*. Plik zawiera kod generujący pozycje *O firmie*, *Oferta*, *Nasze sklepy*. Ustaw utworzony layout jako domyślny dla akcji `index`.

Rozwiązanie

Na początek utwórz kopię layoutu *wizytowki.php* i zmień jej nazwę tak jak w poleceniu. Operacja jest bardzo prosta. Pamiętaj jedynie o tym, że kopia musi być zapisana w katalogu *templates* aplikacji *widok*. W tym samym katalogu utwórz plik *menu.php* i wprowadź do niego poniższy kod:

```
<ul class="fl">
  <li style="background: none;"><a href=""><span>O firmie</span></a></li>
  <li><a href=""><span>Oferta</span></a></li>
  <li><a href=""><span>Nasze sklepy</span></a></li>
  <li><a href=""><span>Referencje</span></a></li>
</ul>
```

Następnie odszukaj w pliku *wizytowki_dolaczenie.php* kod wyglądający bardzo podobnie. Zastąp ten kod (łącznie ze znacznikami `ul`) poniższą instrukcją:

```
<?php include( sfConfig::get( 'sf_app_template_dir' ).'/menu.php' ); ?>
```

Dołączenie pliku wymaga określenia ścieżki do pliku. Z tego powodu musiałeś użyć klasy `sfConfig`. Za pomocą jej metod możesz pobierać informacje o środowisku Symfony. W przykładzie użyłeś metody `get` do pobrania wartości dyrektywy `sf_app_template_dir`. Dyrektywa przechowuje ścieżkę do katalogu z layoutami aplikacji. Należało więc jedynie dokleić do otrzymanej ścieżki nazwę dołączanego pliku, co uczyniłeś.

Wróćmy do przykładu. Plik layoutu po zmianach powinien wyglądać tak:

```
<div id="right_menu">
  <?php echo image_tag('wizytowki/right_menu_top.jpg', 'class="fl"') ?>
  <?php include( sfConfig::get( 'sf_app_template_dir' ).'/menu.php' ); ?>
  <?php echo image_tag('wizytowki/right_menu_bottom.jpg', 'class="fl"') ?>
  <div class="cl"></div>
</div>
```

Kolejny krok, jaki musisz wykonać, polega na ustawieniu layoutu dla akcji `index`. W tym celu otwórz plik *actions.class.php* i zmodyfikuj akcję `index`, jak przedstawiono na listingu:

```
public function executeIndex() {
  $this->getResponse()->addStyleSheet('wizytowki');
  $this->setLayout('wizytowki_dolaczanie');
}
```

Obie instrukcje użyte w akcji poznałeś już wcześniej. Pierwsza dodaje do odpowiedzi plik ze stylami, druga zaś zmienia szablon. Ponieważ wszystko, co trzeba, już wykonałeś, nadszedł czas na testy. W przeglądarce wpisz adres:

http://127.0.0.1/widok.php/elementy_widoku

Jeżeli wszystko zrobiłeś poprawnie, na ekranie zobaczysz layout zawierający menu z pliku *menu.php*.

Niestety, przedstawione rozwiązanie ma kilka istotnych wad. Do najważniejszych z nich należą:

- Zmienne w różnych plikach mogą mieć takie same nazwy. W ten sposób powodujemy konflikt nazw.
- System pamięci podręcznej zaimplementowany w Symfony nie potrafi wykrywać dołączania plików. Z tego powodu każde użycie *menu.php* zostanie wpisane do pamięci podręcznej, co z pewnością nie jest pożądane.

W dalszej części książki poznasz rozwiązania omijające opisane powyżej niedogodności.

Partiale

Partiale są elementami widoku przypominającymi funkcje w języku programowania. Za ich pomocą można grupować kod widoku, a następnie wielokrotnie z niego korzystać. Dla przykładu, kod HTML wyświetlający wiersz tabeli możemy umieścić w partialu.

Następnie partial wywołamy w widoku tyle razy, ile potrzeba. W ten sposób zmiany dotyczące tabeli, takie jak przestawienie kolumn itp., wykonywane są w jednym miejscu. Ponadto partiale rozwiązują problemy opisane w poprzedniej sekcji.

Partiale mogą być używane w różnych elementach widoku. Możemy z nich korzystać w layoutach, szablonach widoku akcji, jak również w innych partialach. Jak rozpoznać partial? Otóż każdy partial musi być zapisany w osobnym pliku. Nazwy tych plików musi rozpoczynać znak podkreślenia. Rozpoznanie partiala nie jest więc trudne.

Ćwiczenie 4.10

Utwórz partial pozwalający wyświetlić na stronie dodatkowe menu. Nowe menu zawiera następujące linki: ostatnio dodane, najpopularniejsze i najlepiej ocenione. Menu ma być pokazane pod utworzonym w ćwiczeniu 4.9.

Rozwiązanie

Partial, który utworzysz, będzie częścią layoutu. Z tego powodu musisz go zapisać w katalogu *templates* dla aplikacji. Przejdź więc do tego katalogu, a następnie utwórz plik o nazwie *_menu_dolne.php*. Poniżej znajduje się przykładowy listing pliku:

```
<ul class="fl">
  <li><a href=""><span>Ostatnio dodane</span></a></li>
  <li><a href=""><span>Najpopularniejsze </span></a></li>
  <li><a href=""><span>Najlepiej ocenione</span></a></li>
</ul>
```

Jak widzisz, jest to menu w takim samym stylu jak poprzednie. Teraz wystarczy dołączyć partial do layoutu. W tym celu znajdź znacznik z atrybutem `id` równym `right_menu`. Po poleceniu

```
<div class="cl"></div>
```

musisz dodać następujący kod:

```
<?php echo image_tag('wizytowki/right_menu_top.jpg', 'class="fl"') ?>
<?php include_partial('global/menu_dolne'); ?>
<?php echo image_tag('wizytowki/right_menu_bottom.jpg', 'class="fl"') ?>
<div class="cl"></div>
```

Zwróć uwagę, w jaki sposób kod partiala umieszczany jest w layoucie. Po pierwsze, musisz wywołać funkcję `include_partial`, która odpowiada za przetworzenie partiala. Po drugie, partiale utworzone dla layoutu należy poprzedzić słowem kluczowym `global`. W ten sposób informujesz Symfony, że potrzebny plik znajdzie w katalogu *templates* aplikacji.

Partiale w widokach akcji

W poprzedniej sekcji dowiedziałeś się, w jaki sposób dołącza się partial zapisany w tym samym katalogu co layout. Tym razem poznasz reguły dołączania partiali w widokach akcji. Na początek wyobraź sobie, że chcesz dołączyć partial `_menu_osobiste` do widoku `logowanieSuccess`. Widok znajduje się w module `uzytkownik`. Poniżej znajdziesz wyjaśnienie, jak możesz to zrobić:

1. Jeżeli partial znajduje się w katalogu *templates* dla modułu `uzytkownik`, dołączysz go za pomocą instrukcji:

   ```
   include_partial('menu_osobite');
   ```

2. Jeżeli partial znajduje się w katalogu *templates* innego modułu, dołączysz go za pomocą instrukcji

   ```
   include_partial('nazwa_modułu/menu_osobiste')
   ```

3. Jeżeli partial znajduje się w katalogu *templates* całej aplikacji, dołączysz go za pomocą instrukcji

   ```
   include_partial('global/menu_osobiste')
   ```

Zagadnienie omówione powyżej sprawdzisz, wykonując kolejne ćwiczenie.

Ćwiczenie 4.11

W module `elementy_widoku` utwórz pustą akcję zamówienia. Do widoku akcji dołącz partial `oferta`. Partial wyświetli ofertę w formie listy.

Rozwiązanie

Zacznij od przygotowań wstępnych. Po pierwsze, utwórz akcję `zamowienia` oraz widok `zamowieniaSuccess`. Po drugie, ustaw dla widoku akcji layout `wizytowki_dolaczanie`. Po trzecie, wyczyść pamięć podręczną i sprawdź, czy widok akcji dekorowany jest odpowiednim layoutem.

Przejdź do katalogu *<nazwa_projektu>/apps/widok/modules/elementy_widoku/templates*. Utwórz nowy plik i nazwij go *_oferta.php*. W pliku umieść kod przedstawiony na listingu:

```
<ul>
  <li>wizytówki</li>
  <li>kalendarze</li>
  <li>prospekty</li>
</ul>
```

Jak widzisz, kod zawiera standardową listę HTML.

W kolejnym kroku musisz wprowadzić zawartość do pliku *zamowieniaSuccess.php*. Przykładowy kod przedstawiony jest poniżej:

```
<p class="header">
  Zamówienia indywidualne
  <?php echo image_tag('wizytowki/ul_dot.gif', 'class="fl"') ?>
  Rok 2006
</p>
<p class="hr"></p>
<p>Treść akcji</p>
<p>Treść akcji</p>
<p>Treść akcji</p>
<p>Treść akcji</p>
<p>Treść akcji</p>
<?php include_partial('oferta'); ?>
```

W widoku akcji użyłeś instrukcji `include_partial`. Przyjmuje ona jeden argument o wartości oferta. W ten sposób lista zostanie dodana do ogólnego wyniku. Ale czy aby na pewno? W przeglądarce wywołaj żądanie:

http://127.0.0.1/widok.php/elementy_widoku/zamowienia

Niestety, widoczny rezultat jest daleki od oczekiwanego. Co więc poszło nie tak? Otóż żeby zobaczyć dane produkowane w widoku akcji, należy na to przygotować layout. W tym celu otwórz go do edycji i znajdź znacznik:

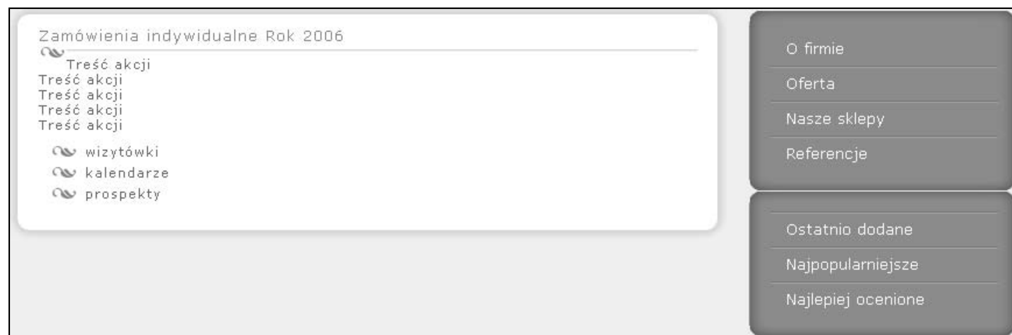
```
<div class="content">
```

Usuń całą zwartość znacznika. Teraz musisz dodać instrukcję wyświetlającą dane z akcji. Poniżej znajduje się kod, który wykona to zadanie:

```
<div class="content">
  <?php echo $sf_data->getRaw('sf_content'); ?>
</div>
```

Zwróć uwagę na obiekt `$sf_data`, który przechowuje rozmaite dane. Można je pobierać, używając metody `getRaw`. Przyjmuje ona jako parametr wartość identyfikującą dane, które chcemy pobrać. Jeżeli użyjesz `sf_content`, pobierzesz kod HTML wyprodukowany przez widok akcji.

Teraz możesz już odświeżyć okno przeglądarki. Wynik akcji powinien być taki jak na rysunku 4.3. Zawiera on widok akcji z kodem partiala opakowany w layout. O to właśnie chodziło.



Rysunek 4.3. *Dekorowanie wyniku akcji layoutem*

Przekazywanie parametrów do partiala

Niewątpliwie olbrzymią zaletą partiali jest możliwość przekazywania dla nich parametrów. W jaki sposób można to zrobić? Należy użyć drugiego parametru, wywołując `include_partial`. Więcej dowiesz się z kolejnego przykładu.

Ćwiczenie 4.12

Utwórz partial `lista`. Jako parametr przyjmuje on tablicę elementów, jakie należy umieścić na liście.

Rozwiązanie

Zadanie, które musisz teraz wykonać, jest nieco bardziej złożone niż poprzednie. Postaram się jednak, aby nie sprawiło Ci zbyt dużych trudności. Zacznij od utworzenia w katalogu *templates* pustego pliku *_lista.php*. Poniżej znajdziesz przykładowy kod partiala:

```
<ul>
  <?php foreach( $aTablicaElementow as $sElement): ?>
    <li><?php echo $sElement; ?></li>
  <?php endforeach; ?>
</ul>
```

W pierwszej chwili Twoją uwagę może przykuć dziwna konstrukcja `foreach`. W przykładzie użyłem alternatywnej składni, która jest bardzo wygodna na użytek szablonów. Pozwala łatwo oddzielić kod HTML od kodu PHP. Więcej na ten temat znajdziesz na stronie *http://php.net.pl/manual/pl/control-structures.alternative-syntax.php*.

Skup się teraz na tym, co jest istotą ćwiczenia. W pętli foreach wykorzystałeś zmienną `$aTablicaElementow`. Skąd ona się tam wzięła? Najprościej rzecz ujmując, zmienna została przekazana jako parametr funkcji `include_partial`. Łatwo zauważysz, że wewnątrz partiala nie trzeba używać żadnej specjalnej konstrukcji.

W kolejnym kroku musisz przygotować dane dla partiala. Przejdź do akcji `zamowienia` i dodaj do niej następujący kod:

```
$this->aOferta = array(
  'wizytówki',
  'kalendarze',
  'prospekty'
);
```

Powyższy kod jest niczym więcej jak tablicą zawierającą elementy o wartości: `wizytówki`, `kalendarze` i `prospekty`. Zwróć uwagę na to, że tablica będzie w widoku akcji dostępna pod nazwą `$aOferta`.

Nadszedł czas na wypróbowanie możliwości partiala. Dodaj na końcu widoku `zamowie↳niaSuccess` kod przedstawiony na listingu:

```
<p class="hr"></p>
<?php include_partial( 'lista', array( 'aTablicaElementow' => $aOferta) ); ?>
```

Jak widzisz, parametry do partiala przekazywane są za pomocą tablicy asocjacyjnej. Bardzo ważne jest, żebyś zapamiętał, że indeks w tablicy asocjacyjnej jest nazwą zmiennej w partialu. Innymi słowy, tablica `$aOferta` będzie w partialu dostępna pod nazwą `aTablicaElementow`.

Na koniec odśwież żądanie w oknie przeglądarki. Jeżeli wszystko wykonałeś poprawnie, na ekranie zobaczysz listę wyglądającą tak samo jak wygenerowana przez partial `oferta`. Zasadnicza różnica polega na tym, że partial `lista` może generować dowolnie wiele różnych list, podczas gdy partial `oferta` zawsze będzie taki sam.

Krótkie podsumowanie

W dwóch ostatnich ćwiczeniach poznałeś już nieco partiale. Spróbuję krótko podsumować najważniejsze aspekty omawianych zagadnień:

- Partiale można dołączać do dowolnego widoku akcji. Z tego względu nie mogą one bezpośrednio korzystać ze zmiennych widoków.
- Zmienne potrzebne w partialach należy przekazywać za pomocą tablicy parametrów.
- Każdy partial ma własną przestrzeń nazw, więc nie musimy się martwić o nazwy zmiennych.
- Mechanizmy pamięci podręcznej Symfony potrafią wykrywać partiale. Stąd wynika, że potrafią również zapisywać je w keszu.
- Parametry dla partiala przekazywane są w postaci tablicy asocjacyjnej. Dostęp do nich uzyskujemy jednak tak jak przy zmiennej skalarnej. Nazwa zmiennej skalarnej jest taka jak nazwa indeksu w tablicy.

Komponenty

Na początku podrozdziału warto zadać pytanie: Czym są komponenty? Kiedyś usłyszałem dosyć ciekawą definicję: „komponent to taki trochę mądrzejszy partial". W tak przedstawionej definicji jest sporo prawdy, ponieważ komponentu używa się wtedy, kiedy partial wymaga pewnej logiki. Zapamiętaj więc bardzo ważną regułę: **jeżeli partial robi się zbyt złożony i zawiera pewne elementy logiki, musisz przekształcić go w komponent**. W tej części nauczysz się, jak to zrobić!

Komponent w Symfony jest bardzo podobny do akcji. Zasadnicza różnica polega na szybkości działania. Komponenty są znacznie szybsze, co jest dodatkowym argumentem przemawiającym za tym, aby rozważyć ich użycie tam, gdzie jest taka możliwość. Zanim przejdziemy dalej, musisz poznać kilka konwencji:

- Nazwa pliku zawierającego komponenty to *components.class.php*. Plik musi być zapisany w katalogu *actions*.
- Klasą bazową dla komponentów jest klasa `sfComponents`.
- Metoda zawierająca kod komponentu nazywana jest według schematu `executeNazwaKomponentu`.
- Komponent musi być połączony z partialem, który odpowiada za jego wizualizację. Plik partiala nazywamy według schematu *_NazwaKomponentu.php*.

Komponenty możemy umieszczać w oddzielnych plikach. Zasady są takie same jak dla pojedynczych akcji.

Ćwiczenie 4.13

Utwórz komponent, którego zadanie będzie polegało na wyświetlaniu losowej sentencji.

Rozwiązanie

W katalogu *actions* (modułu `elementy_widoku`) utwórz pusty plik *sentencjeComponent.class.php*. Poniżej listing zawartości pliku:

```
class sentencjeComponent extends sfComponent {

  public function execute() {
    $aSentencje = array(
  'Chciano za szczęściem rozpisać listy gończe, ale nikt nie umiał podać rysopisu.
     Wiesław Brudziński',
  'Nie ma większej radości dla głupiego, jak znaleźć głupszego od siebie.
     Aleksander Fredro',
      'Nie możesz iść do przodu, patrząc w tył. Kostis Palamas',
      'Życie jest jak tramwaj, nadmiar bagażu utrudnia ci jazdę. Antoni
      ↪Uniechowski',
      'Nie wierzcie zegarom, czas wymyślił je dla zabawy. Wojciech Bartoszewski',
      'W życiu można liczyć tylko na siebie, ale i tego nie radzę. Tristan Bernard'
    );
    $iLosowa = rand(0, count($aSentencje) - 1);
    $this->sSentencjaNaDzis = $aSentencje[$iLosowa];
  }

}
```

Analizę przykładu rozpocznij od nagłówka klasy. Została ona nazwana `sentencjeCompo↪nent` i wyprowadzona z klasy `sfComponent` (klasa bazowa dla pojedynczych akcji). Symfony wymusza na Tobie stosowanie odpowiednich konwencji. Jeżeli chcesz utworzyć pojedynczy komponent sentencje, musisz zapisać go w pliku o nazwie *sencje Component.class.php*, klasę zaś musisz nazwać `sentencjeComponent`.

Dalsza część kodu jest raczej łatwa do zrozumienia. Tworzysz tablicę sentencji. Następnie losujesz numerek dla sentencji, którą należy pokazać w widoku. Wylosowaną sentencję przypisujesz do zmiennej, która w widoku będzie dostępna pod nazwą `$sSentencjaNa↪Dzis`. Warto zauważyć w tym miejscu, że w normalnych warunkach sentencja mogłaby być losowana z bazy danych.

W kolejnym kroku musisz utworzyć partial, który będzie powiązany z komponentem. Zadanie jest naprawdę proste. W katalogu *templates* utwórz plik o nazwie *_sentencje.php*. Poniżej znajdziesz jego zawartość:

```
<div><?php echo $sSentencjaNaDzis?></div>
```

Prawda, że proste?! Powyższy kod wymaga tylko jednego komentarza. W tego typu partialach możliwy jest dostęp do zmiennych komponentu — oczywiście do zmiennych poprzedzonych `$this`. Czy nie jest to znajome? Otóż, komponent do swojego partiala ma się tak jak akcja do swojego widoku.

Ostatnie zadanie polega na wywołaniu komponentu w szablonie zamowieniaSuccess. Początkowy kod:

```
<p class="header">
  Zamówienia indywidualne
  <?php echo image_tag('wizytowki/ul_dot.gif', 'class="fl"') ?>
  Rok 2006
</p>
```

musisz zastąpić poniższym:

```
<p class="header">
  <?php include_component( 'elementy_widoku', 'sentencje' ); ?>
</p>
```

Za pomocą instrukcji include_component dołączamy do szablonu kod wygenerowany przez komponent. Zwróć uwagę na to, że include_component wymaga dwóch argumentów. Pierwszy jest nazwą modułu, w którym zapisany jest komponent, drugi zaś nazwą komponentu, który należy dołączyć.

Na koniec przejdź do przeglądarki i odśwież kilka razy zawartość okna. Sentencja na początku okna z treścią powinna się losowo zmieniać.

> Komponent to mądrzejsza wersja partiala, więc możesz również do niego przekazywać parametry za pomocą tablicy. Należy ją przekazać do funkcji include_component jako ostatni argument. Każda zmienna przekazana w ten sposób dostępna jest poprzez $this->{*nazwa_zmiennej*}.

Krótkie podsumowanie

Na początku komponenty mogą Ci się wydać nieco dziwne. Zapewniam jednak, że po krótkim czasie korzystania z nich docenisz ich zalety. Poniżej znajdziesz kilka bardzo ważnych reguł dotyczących komponentów:

- Komponent dodawany jest za pomocą instrukcji include_component. Wymaga ona dwóch parametrów: nazwy modułu i nazwy komponentu.
- Zachowanie konwencji nazewniczej pozwala komponentowi użyć właściwego partiala do wizualizacji wyników.
- Do partiala związanego z komponentem można przekazywać parametry za pomocą $this.
- Komponent ma dostęp do tych samych obiektów globalnych co akcja.
- Komponenty nie mają sekcji bezpieczeństwa oraz metod walidacji (jest to powód ich szybkości).
- Komponenty nie mogą być wywoływane za pomocą żądania, a jedynie przez aplikacje (w przeciwieństwie do akcji).

We wszystkich przykładach prezentowanych w poprzedniej części rozdziału stosowałeś include_partial lub include_component. Działanie ich powoduje wysłanie na wyjście kodu partiala lub komponentu. Nie zawsze takie zachowanie jest pożądane. Wyobraź sobie sytuację, w której partial odpowiada za wygenerowanie formularza. W dalszej części formularz może być umieszczony w jednym z kilku specjalnie do tego wystylizowanych elementów strony (element szukaj, logowanie itp.). Wynika stąd, że formularz nie powinien być od razu wysyłany na wyjście, tylko przechowywany w jakiejś zmiennej do czasu, kiedy aplikacja podejmie decyzję, w którym elemencie strony należy go umieścić. Mam nadzieję, że jest to dla Ciebie zrozumiałe.

Alternatywą dla używania include_partial i include_component są funkcje get_partial i get_component. Nie powodują one wysyłania na wyjście kodu utworzonego przez partial. Zamiast tego zwracają kod jako wyjście funkcji.

Sloty

Przypomnij sobie sytuację, kiedy w jednym z ćwiczeń nie widać było treści akcji w layoucie. Poznałeś wówczas obiekt $sf_data oraz jego metodę getRaw. Metoda ta wymaga jednego parametru, który określa, jakie dane chcesz wyświetlić. Co jednak w sytuacji, kiedy takich miejsc na stronie jest więcej?

Koncepcja slotów pozwala zafiksować w layoucie pewne miejsca i przypisać do nich nazwę. Dane produkowane przez akcje mogą być przekazywane do jednego z takich miejsc (slotu). Ponieważ każdy slot ma własną, unikatową nazwę, dane zawsze trafią we właściwe miejsce.

Możesz traktować slot jako pewną zmienną globalną, dla której dane generowane są przez partiale i komponenty. Wartość przeznaczona dla slotu utrzymywana jest globalnie, więc może być określana w dowolnym elemencie widoku. Przy czym ważne jest, żeby pamiętać o tym, że:

- layout jest generowany na końcu (wzorzec dekoratora),
- partiale i komponenty wykonywane są w miejscach, w których się do nich odwołujemy.

Z powyższych punktów wynika, że layout zawsze otrzymuje dla slotów najbardziej aktualne dane.

Ćwiczenie 4.14

Utwórz moduł o nazwie sloty oraz layout o nazwie wizytowki_sloty. Layout musi być domyślny dla całego modułu. Ponadto dodaj do layoutu dwa sloty: na treść oraz na pasek nawigacyjny. Działanie slotów przetestuj za pomocą akcji zamowienie.

Rozwiązanie

Utworzenie modułu oraz layoutu pozostawiam Tobie jako ćwiczenie powtórzeniowe. Również konfigurację domyślnego layoutu potraktuj jako odświeżenie wcześniej poznanych wiadomości.

Zacznij od dodania do nowego layoutu wymaganych slotów. Odszukaj w kodzie linijkę:

```
<p class="bc"><a href="">Strona główna</a> / <a href="">Zamówienia indywidualne</
↳a> / <a href="">Rok 2006</a></p>
```

i zamień ją na:

```
<p class="bc">
  <?php include_slot( 'nawigator' ); ?>
</p>
```

W ten sposób umieściłeś w layoucie slot o nazwie `nawigator`. W dalszej części dowiesz się, w jaki sposób akcje mogą umieszczać w tym slocie swoje dane.

Kolej na drugi slot. Znajdź kod:

```
<?php echo $sf_data->getRaw('sf_content'); ?>
```

i zamień go na:

```
<?php include_slot( 'tresc_akcji' ); ?>
```

Teraz musisz sprawdzić, czy sloty działają. W tym celu utwórz w module `slot` akcję o nazwie `zamowienia`. Umieść w jej widoku poniższy kod:

```
<p class="header">
  <?php include_component( 'elementy_widoku', 'sentencje' ); ?>
</p>
<p class="hr"></p>
<p>Treść akcji</p>
<p>Treść akcji</p>
<p>Treść akcji</p>
<p>Treść akcji</p>
<p>Treść akcji</p>
<?php include_partial('elementy_widoku/oferta'); ?>
<p class="hr"></p>
<?php
  include_partial( 'elementy_widoku/lista', array( 'aTablicaElementow' => $aOferta) );
?>
```

Zwróć uwagę na sposób wykorzystania partiala napisanego w części modułu `elementy_`
`↳widoku`. Zgodnie z tym, co mogłeś przeczytać wcześniej, nazwę modułu podajemy przed nazwą partiala. Ponadto do partiala przekazywany jest parametr `aTablicaElementow`. Zwróć uwagę, że zmienna `$aOferta`, jaką jest on inicjalizowany, nie istnieje. Z tego powodu dodaj do akcji `zamowienia` poniższy kod:

```
$this->aOferta = array(
  'wizytówki',
  'kalendarze',
  'prospekty'
);
```

Teraz możesz już wywołać żądanie, wprowadzając do przeglądarki adres:

http://127.0.0.1/widok.php/sloty/zamowienia

Jaki to daje efekt? Daleki od oczekiwanego. Co więc jest nie tak? Otóż dodałeś do layoutu sloty, ale nie umieściłeś o tym w akcji ani słowa. Teraz musisz to zmienić. W pierwszej linijce widoku `zamowieniaSuccess` dodaj instrukcję:

```
<?php slot('tresc_akcji'); ?>
```

Instrukcja informuje Symfony, że wszystkie dane od tej pory należą do slotu o nazwie `tresc_akcji`. W ostatniej linijce widoku dodaj:

```
<?php end_slot(); ?>
```

Ta instrukcja informuje Symfony, że zapisywanie danych do slotu należy zakończyć.

Spróbuj ponownie uruchomić żądanie. I jak poszło tym razem? Lepiej? O wiele! Brakuje jednak jeszcze jednego elementu. Paska nawigacyjnego. Na początku widoku akcji dodaj poniższy kod:

```
<?php slot('nawigator'); ?>
  <a href="">Strona główna</a> /
  <a href="">Zamówienia indywidualne</a> /
  <a href="">Rok 2006</a>
<?php end_slot(); ?>
```

Jak widzisz, kod zaczyna się od informacji, że rozpoczynają się dane przeznaczone dla slotu `nawigator`. Potem następuje kilka znaczników hiperłączy, po nich zaś informacja, że są to już wszystkie dane dla slotu. Odśwież stronę i zobacz, czy dane umieszczone są we właściwym miejscu. Bingo!

Na koniec rzuć okiem na całe ćwiczenie. Poznałeś w nim kilka bardzo ważnych tajników Symfony. Po pierwsze, layout może zawierać wiele slotów. Po drugie, w widoku akcji możesz przesyłać dane do tylu slotów, ilu wymaga funkcjonalność serwisu.

Co się dzieje z poleceniami HTML/PHP wywoływanymi w widoku poza obszarem działania slotu? Po wywołaniu żądania nigdy nie będą one widoczne. Potraktuj to jako ćwiczenie do samodzielnego wykonania.

Krótkie podsumowanie

Pracując ze slotami, musisz pamiętać o kilku bardzo istotnych rzeczach. Oto najważniejsze z nich:

- Dane do slotów można przekazywać z każdego elementu widoku (komponent, partial, szablon).
- Kod wykonywany między pomocnikami slotu (`slot`, `end_slot`) ma dostęp do tych samych zmiennych, co element piszący do slotu.
- Wynik slotu umieszczany jest automatycznie w obiekcie odpowiedzi (`response`) i nie jest pokazywany w szablonie.

- Wyświetlanie zwartości slotu następuje dopiero podczas wywołania funkcji `include_slot`, zawierającej jego nazwę.

Zapamiętaj te reguły, ponieważ ułatwią Ci programowanie w Symfony.

Konfiguracja

Tworzenie szablonów widoków uwzględnia fakt, że graficy zwykle nie znają języków programowania. Z tego względu projektanci Symfony wydzielili folder *template* na wszystkie pliki prezentujące dane. Przykładowa struktura aplikacji wygląda więc jak na listingu:

```
projekt/
  apps/
    aplikacja/
    templates/  //layouty dla aplikacji
    modules/
      moduł1/
        templates/  //elementy widoku dla modułu 1
      moduł2/
        templates/  //elementy widoku dla modułu 2
      moduł3/
        templates/  //elementy widoku dla modułu 3
```

Wszystkie elementy widoku można sklasyfikować w dwóch grupach: te, które generują część prezentacyjną, oraz pozostałe. Do pozostałych zalicza się znaczniki meta, tytuł strony, pliki css, pliki js itp. Elementy należące do tej grupy możesz bez problemu konfigurować za pomocą plików *view.yml* oraz obiektu odpowiedzi (jest instancją klasy `sfWebResponse`). W tej sekcji poznasz obie metody.

Pliki view.yml

Aplikacja oraz moduły mogą posiadać własny plik *view.yml*. Plik konfiguracyjny dla aplikacji tworzony jest automatycznie, pliki zaś konfiguracyjne dla modułów musisz utworzyć ręcznie, jeśli ich potrzebujesz. Zwróć uwagę na to, że każdy moduł zawiera pusty katalog *config*. Katalog ten jest właściwym miejscem dla pliku *view.yml* określającego parametry dla modułu. Poniżej ogólny schemat pliku:

```
akcja1Success:
  opcja1: wartość1
  opcja2: wartość2
akcja2Success:
  opcja1: wartość1
  opcja2: wartość2
akcja1Error:
  opcja1: wartość1
```

Na powyższym schemacie możesz zauważyć kilka istotnych prawidłowości. Po pierwsze, parametry konfiguracyjne określa się dla konkretnego szablonu widoku. Po drugie, konfigurować możesz dowolne typy szablonów. Nie ma znaczenia, czy jest to widok sukcesu, błędu, czy może też jakiś inny.

O czym jeszcze warto wiedzieć? Jeżeli chcesz ustawić pewien parametr dla wszystkich szablonów w module, możesz użyć sekcji `all`. Wygląda to mniej więcej tak:

```
all:
  parametr: wartość
```

Jako przykład wykonaj następne ćwiczenie.

Ćwiczenie 4.15

Dla wszystkich widoków w module `sloty` zmień tytuł strony na *Sloty Wizytówki*.

Rozwiązanie

Otwórz do edycji plik *view.yml* zapisany w katalogu *config* należącym do modułu `sloty`. Do sekcji all dodaj poniższe linijki:

```
metas:
  title: Sloty Wizytówki
```

Pamiętaj o spacjach oraz o wyczyszczeniu keszu. Przetestuj wynik w przeglądarce.

Kaskada plików konfiguracyjnych

Jak widzisz, plików konfiguracyjnych może być w jednej aplikacji całkiem sporo. Zasadne jest więc pytanie, jaki będzie efekt, jeżeli ta sama opcja zostanie zadeklarowana w różnych miejscach? Żeby to zrozumieć, musisz wiedzieć, w jakiej kolejności Symfony przetwarza pliki oraz dyrektywy.

1. Wczytywany jest plik z katalogu *config* dla całej aplikacji.
2. Wczytywany jest plik z katalogu *config* umieszczonego w module.
3. Sekcja `all` dla modułu nadpisuje ustawienia z aplikacji.
4. Sekcje dla widoków w pliku modułu nadpisują wcześniej ustawione wartości w pliku aplikacji oraz sekcji `all`.

Reguły są więc jasne. Jeżeli masz jeszcze jakieś wątpliwości, przeczytaj poniższy przykład.

Załóżmy, że masz dwa pliki konfiguracyjne: jeden dla aplikacji i jeden dla modułu. Plik dla aplikacji zawiera poniższe dyrektywy:

```
default:
  metas:
    title:        Superwizytówki na każdą kieszeń
    description:  Tanie wizytówki dla firm.
```

```
    keywords:     wizytówki, tanie wizytówki
    language:     pl
```

Plik modułu również określa część tych samych dyrektyw i jest przedstawiony poniżej:

```
all:
  metas:
    title:         Wizytówki dla Ciebie
    description:   Wizytówki na każdą okazję.
ofertaSuccess:
  metas:
    title:       Wizytówki w naszej ofercie
```

Pytanie, które należy teraz zadać, brzmi: jakie będą wartości parametrów `title`, `description` oraz `language` dla widoku akcji `oferta`? Na początek przyjrzyj się parametrowi `title`. Określasz wartość dla aplikacji, która następnie zostaje nadpisana przez wartość podaną w sekcji `all` modułu. Na koniec dyrektywa jest jeszcze nadpisana przez parametry określone dla widoku `ofertaSuccess`. Wniosek nie może być inny: wartość parametru `title` jest równa `"Wizytówki w naszej ofercie"`.

Taką samą analizę przeprowadź teraz dla dyrektywy `description`. Aplikacja ustawia dla niej wartość `"Tanie wizytówki dla firm."`. Następnie wartość nadpisywana jest przez moduł. Ponieważ w sekcji `ofertaSuccess` nie określono wartości dla description, wartość końcowa parametru jest równa `"Wizytówki na każdą okazję."`.

Na koniec parametr `language`. Jest on ustawiany tylko przez aplikację, więc jego wartość we wszystkich widokach modułu jest równa `pl`. Również w widoku akcji `oferta`.

Obiekt Response

Podstawowa różnica między plikiem *view.yml* a obiektem odpowiedzi jest taka, że obiektu można używać w akcjach. Czasami ułatwia to znacznie proces konfiguracji oraz zdecydowanie rozszerza jej możliwości.

Na co pozwala obiekt odpowiedzi? Zastosowań ma bardzo wiele. Za jego pomocą możesz zmienić tytuł strony, zawartość znaczników meta, ustawić ciasteczka itp. Poznasz go lepiej, realizując kolejne zadania.

Na początek utwórz w aplikacji `widok` moduł `response`. Wyłącz dekorowanie layoutem dla wszystkich szablonów widoku w module.

Ćwiczenie 4.16

Utwórz akcję `ciasteczko`. Zadaniem akcji jest przesłanie do klienta ciasteczka `moje_ciasteczko`. Następnie sprawdź za pomocą akcji `pobierzCiasteczko`, czy operacja została wykonana poprawnie.

Rozwiązanie

Zacznij od akcji `ciasteczko`. Poniżej znajdziesz przykładowy kod:

```
public function executeCiasteczko() {
  $oResponse = $this->getResponse();
  $oResponse->setCookie('moje_ciasteczko','szarlotka_babci', time()+3600);
  return sfView::HEADER_ONLY;
}
```

Na początku pobierasz referencję do obiektu `Response`. Za pomocą metody `setCookie` ustawiasz nazwę ciasteczka, jego wartość oraz termin ważności. Na koniec informujesz, że akcja przesyła jedynie nagłówki i nie potrzebuje widoku.

Kolej na pobieranie ciasteczka. Wprowadź do pliku akcji poniższy kod:

```
public function executePobierzCiasteczko() {
  $this->sCiasteczko = $this->getRequest()->getCookie('moje_ciasteczko');
}
```

Ciasteczko przesyłane jest przez przeglądarkę w żądaniu klienta. Z tego względu dostęp do niego uzyskujesz za pomocą obiektu żądania (`Request`). Jak widzisz, wystarczy jedynie skorzystać z metody `getCookie` i po sprawie.

Musisz już tylko utworzyć widok dla akcji `pobierzCiasteczko`. Może on wyglądać jak na listingu poniżej:

```
<?php
echo 'pyszne to ',$sCiasteczko,'<br />';
?>
```

Jak widzisz, jest on bardzo prosty.

Na koniec musisz wykonać testy. W tym celu wywołaj żądanie:

http://127.0.0.1/widok.php/response/ciasteczko

Następnie sprawdź wynik działania pod adresem:

http://127.0.0.1/widok.php/response/pobierzCiasteczko

Na ekranie powinieneś zobaczyć wartość ciasteczka.

Wskazówka

> Pamiętaj, że ciasteczko nie jest widoczne w żądaniu, które wywołuje metodę `set↪Cookie`. To żądanie przesyła do przeglądarki informację, że należy je utworzyć. Dopiero przy kolejnym żądaniu przeglądarka uwzględni ciasteczko i prześle je do serwera.

Sterowanie sekcją meta poprzez obiekt odpowiedzi

Obiekt odpowiedzi umożliwia również dodawanie wpisów dla sekcji meta. Takie podejście ma co najmniej dwie zalety. Po pierwsze, nie wymaga czyszczenia pamięci podręcznej. Po drugie, konfigurację określasz za jednym zamachem dla wszystkich widoków akcji. Może być to bardzo przydatne.

Ćwiczenie 4.17

Utwórz akcję `meta` (pamiętaj o pustym widoku). Dla akcji ustaw layout `wizytowki_sloty` oraz określ wszystkie dodatkowe parametr takie jak pliki CSS, tytuł itp.

Rozwiązanie

Zadanie musisz zrealizować, korzystając z klasy `sfWebResponse`. Poniżej proponowany kod akcji:

```
public function executeMeta() {
  $this->setLayout('wizytowki_sloty');

  $oResponse = $this->getResponse();
  $oResponse->addMeta('robots', 'NONE');
  $oResponse->addMeta('keywords', 'wizytówki');
  $oResponse->setTitle('Wizytówki na każdą kieszeń');
  $oResponse->addStyleSheet('wizytowki.css');
  $oResponse->addJavaScript('brak.js');
}
```

Na początku za pomocą `setLayout` określasz, jak powinien być dekorowany kod akcji. W następnych instrukcjach pobierasz referencję do obiektu odpowiedzi i przy jej użyciu określasz słowa kluczowe (*keywords*), sposób indeksacji strony w wyszukiwarkach (*robots*), tutuł oraz dodajesz pliki zewnętrzne: *wizytowki.css* i *brak.js*.

Wpisz w pasku adresu tekst:

http://127.0.0.1/widok.php/response/meta

Jeżeli na ekranie pojawi się layout, oznacza to, że zadanie wykonałeś poprawnie.

O czym musisz jeszcze wiedzieć? Metody obiektu odpowiedzi nadpisują ustawienia plików konfiguracyjnych. Nagłówki odpowiedzi (na przykład ciasteczka) są przysyłane tak późno, jak to tylko możliwe.

> Wszystkie metody `set` mają swoje odpowiedniki `get`. Zwracają one bieżącą wartość danego elementu.

Pliki zewnętrzne

Praktycznie każda strona korzysta z zewnętrznych plików CSS. Niektóre strony (jest ich coraz więcej) korzystają z plików JS. Musisz więc wiedzieć, jak poradzić sobie w sytuacji, kiedy takie pliki trzeba dołączyć do stron projektu.

Na początek utwórz w akcji `widok` moduł o nazwie `pliki_zewnetrzne`. W tym module będziesz wykonywał wszystkie bieżące ćwiczenia.

Pliki CSS i JS

Na początek proponuję krótkie przypomnienie. We wcześniejszych sekcjach tego rozdziału dołączyłeś do wyniku pliki CSS za pomocą pliku *view.yml* (sekcja `stylesheets`) oraz obiektu `Response`. W jednym z ostatnich przykładów dołączałeś również pliki JS. Różnica w dołączaniu polega jedynie na tym, że dla plików JavaScript używasz sekcji `javascripts` oraz metody `addJavaScript`.

Pamiętaj, że pliki ze stylami i skryptami muszą być dostępne publicznie. Z tego względu musisz je umieszczać w katalogu *web/css* oraz *web/js*.

Dołączanie plików nie jest jedynym zadaniem, przed jakim staniesz, realizując komercyjne projekty. Bardzo często będziesz musiał pewne pliki usunąć po to, żeby nie kolidowały z innymi. Do takich kolizji możemy zaliczyć nadpisywanie reguł CSS, zadeklarowanie funkcji o tej samej nazwie itp.

Przyjrzyj się poniższemu plikowi *view.yml*:

```
all:
  layout: sezonowy_slot
  stylesheets: [wizytowki, s1, s2, s3, s4]
```

Wszystkie akcje w module będą dekorowane layoutem `sezonowy_slot`. Ponadto do wyniku zostaną dołączone cztery pliki CSS o nazwach *wizytowki*, *s1*, *s2*, *s3* i *s4*. Na razie wszystko jest w jak najlepszym porządku. Załóżmy jednak, że w module jest akcja o nazwie `specjalna`, która potrzebuje dodatkowego pliku *s5*. Niestety, plik ten koliduje z plikami *s1* i *s3*. Co w takim wypadku należałoby zrobić? Zerknij na poniższą konfigurację:

```
all:
  layout: sezonowy_slot
  stylesheets: [wizytowki, s1, s2, s3, s4]
specjalnaSuccess:
  stylesheets: [-s1, -s3, s5]
```

Okazuje się, że wykluczanie plików z widoku akcji nie jest zbyt trudne. Wystarczy przed nazwą, którą chcesz wykluczyć, dodać znak minus.

Manipulowanie kolejnością dołączanych plików

Z punktu widzenia reguł CSS kolejność plików ma ogromne znaczenie. Reguły, które występują w najdalej (względem początku kodu) dodanych plikach nadpisują te, które były wcześniej (poczytaj o kaskadzie stylów). Wychodząc naprzeciw oczekiwaniom użytkownika, Symfony dostarcza pewnych narzędzi do manipulowania kolejnością plików.

Ćwiczenie 4.18

Dodaj do modułu akcję `kolejnosc_plikow`. Dla akcji wyklucz styl *s1*, dodaj na początek listy plik *pierwszy* oraz na koniec plik *ostatni*.

Rozwiązanie

Dopisz do pliku *view.yml* poniższe dyrektywy:

```
kolejnoscPlikowSuccess:
  stylesheets: [-s1, pierwszy: {position: first}, ostatni: {position: last}]
```

Wnioski powinny nasunąć Ci się nasunąć same. Otóż po nazwie pliku możesz wpisać znak dwukropka. Oznacza on, że wystąpi jeszcze sekcja opcji dla pliku. Sekcja zaznaczana jest za pomocą nawiasów klamrowych. W obu sekcjach użyto tylko jednej opcji o nazwie `position`. Może ona przyjąć dwa parametry:

- `first` — dodaj plik na początku listy,
- `last` — dodaj plik na końcu listy.

Nie jest to doskonałe narzędzie i nadaje się głównie do prostych przetasowań. Bardzo często, kiedy zmiany w kolejności są znaczące, dodaję do pliku konfiguracyjnego wpis podobny do tego poniżej:

```
stylesheets: [-*, pierwszy, wizytowki, s2, s3, s4, ostatni]
```

Za pomocą łańcucha `-*` określasz, że należy usunąć wszystkie pliki z listy, a następnie podajesz prawidłową kolejność plików.

Określanie medium

Musisz pamiętać, że założenia projektu mogą przewidywać stylizację strony dla więcej niż jednego medium. W takiej sytuacji należy utworzyć kilka plików CSS reprezentujących założone w projekcie media. W jaki sposób możesz potem w Symfony określić, który plik reprezentuje jakie medium? Wykonaj poniższe ćwiczenie.

Ćwiczenie 4.19

Z założenia projektu wynika, że strona generowana przez akcję `media` ma być dostępna dla standardowego monitora, drukarki oraz palmtopa. Utwórz w pliku konfiguracyjnym wpis, który pozwoli uzyskać ten efekt.

Rozwiązanie

Otwórz do edycji plik *view.yml* i dodaj do niego poniższe wpisy:

```
mediaSuccess:
  stylesheets: [mon: {media: screen}, druk: {media: print}, palm: {media: palmtop} ]
```

Jak widzisz, w rozwiązaniu również użyto sekcji opcji. Tym razem potrzebna dyrektywa to media. Wykorzystałeś tutaj trzy wartości dla tego parametru:

- `screen` — ekran monitora,
- `print` — drukarkę,
- `palmtop` — palmtop.

Uruchom akcję w okienku przeglądarki, zajrzyj do kodu strony i upewnij się, że media zostały poprawnie dodane. Pamiętaj również o wyczyszczeniu pamięci podręcznej.

Komponenty slotowe

Komponent slotowy jest skrzyżowaniem slotu i komponentu. Pozwala on Tobie na pisanie komponentów, które będą automatycznie wstawiane w pewnym miejscu layoutu czy też innego elementu widoku. Jak do tej pory, brzmi znajomo. W zasadzie opisałem działanie komponentu. Jaka jest więc różnica między komponentem i komponentem slotowym? Otóż komponent slotowy jest komponentem przypisanym do slotu. Można go w dowolny sposób konfigurować. Wyobraź sobie taką sytuację. Masz w aplikacji dwa odrębne moduły. Każdy z nich posiada własne menu. Menu powinno pojawiać się w tym samym miejscu (zamiast poprzedniego). Mógłbyś użyć do wykonania zadania modułu i umieścić w nim logikę generującą poprawny kod. Mógłbyś też utworzyć slot i umieszczać w nim poprawne menu z poziomu szablonu. Oczywiście musiałbyś to zrobić w każdej akcji należącej do modułu. Jedno i drugie rozwiązanie mnie nie zadowala. Wszystkie pozostałe również ;). W tym miejscu zdecydowanie najlepiej byłoby użyć komponentów slotowych. Mechanizm działa mniej więcej tak. Tworzysz dwa komponenty menu. Tworzysz komponentowy slot w layoucie. Na poziomie modułu określasz, który moduł generuje kod dla slotu. Więcej wyjaśnień w poniższym ćwiczeniu. Przed wykonaniem ćwiczenia utwórz moduł `komponent_slotowy`.

Ćwiczenie 4.20

Utwórz komponent slotowy o nazwie `menu_slotowe`. Następnie utwórz dwa komponenty o nazwach `menuProfil` i `menuZalogowany`. Jeżeli użytkownik wywoła akcję `profil`, powinien zobaczyć kod wygenerowany przez `menuProfil`. Z drugiej strony, jeżeli użytkownik wywoła akcję `zalogowany`, powinien zobaczyć kod wygenerowany przez `menuZa` ↪`logowany`.

Rozwiązanie

Na początek sprawy związane z layoutem. Utwórz kopię layoutu `wizytowki_sloty` i nazwij ją `wizytowki_komponent_slotowy`. Skonfiguruj cały moduł `komponent_slotowy` tak, żeby korzystał on z tego właśnie layoutu. Następnie utwórz dwie puste akcje `profil` i `zalogowany`. Dla akcji utwórz również dwa puste szablony sukcesu.

Zadania wstępne zakończone, możesz przystąpić do kolejnych kroków. Na początek utwórz komponent `menuProfil`. Poniżej znajduje się jego kod:

```
class menuProfilComponent extends sfComponent {

  public function execute() {

  $this->aProfil = array(
    array('ustawienia','profil/ustawienia'),
    array('dane osobowe','profil/dane')
  );
  }

}
```

Komponent zawiera jedynie tablicę pozycji menu. Poniżej znajdziesz kod partiala połączonego z tym komponentem:

```
<ul class="fl">
  <?php
    foreach( $aProfil as $aPozycja ) {
      echo '<li>', link_to( '<span>' . $aPozycja[0] . '</span>',
      ➥$aPozycja[1]),'</li>';
    }
  ?>
</ul>
```

Jedynym zadaniem kodu jest wygenerowanie HTML-a obrazującego pozycje menu.

Drugi komponent `menuZalogowany` znajdziesz na poniższym listingu:

```
class menuZalogowanyComponent extends sfComponent {

  public function execute() {
    $this->aZabezpieczone = array(
      array('poczta','zabezpieczone/poczta'),
      array('oceny','zabezpieczone/oceny')
    );
  }

}
```

Jak widzisz, kod jest niemal identyczny. Różni się tylko pozycjami menu. Partial dla tego komponentu przedstawiony jest poniżej:

```
<ul class="fl">
  <?php
    foreach( $aZabezpieczone as $aPozycja ) {
      echo '<li>', link_to( '<span>' . $aPozycja[0] . '</span>',
      ➥$aPozycja[1]),'</li>';
    }
  ?>
</ul>
```

I tym razem nie różni się za bardzo od poprzedniego. Inna jest jedynie tablica, z której pochodzą pozycje.

Kolej na edycję layoutu. Znajdź w kodzie element `div` o identyfikatorze równym `right_`
`➥menu`. Przed jego znacznikiem zamykającym dodaj kod:

```
<?php echo image_tag('wizytowki/right_menu_top.jpg', 'class="fl"') ?>
<?php echo include_component_slot( 'menu_slotowe' ); ?>
<?php echo image_tag('wizytowki/right_menu_bottom.jpg', 'class="fl"') ?>
<div class="cl"></div>
```

Najważniejsza na listingu jest pogrubiona linijka. To właśnie w niej określasz, gdzie ma być dołączony komponent slotowy. Ponadto musi on być konfigurowany przy użyciu dyrektywy `menu_slotowe`. Więcej na ten temat przeczytasz za chwilę.

Utwórz dla modułu plik *view.yml*. Wprowadź do niego poniższy tekst:

```
all:
  layout:  wizytowki_komponent_slotowy
  stylesheets: [wizytowki]
  components:
    menu_slotowe: []
profilSuccess:
  components:
    menu_slotowe: [komponent_slotowy, menuProfil]
zalogowanySuccess:
  components:
    menu_slotowe: [komponent_slotowy, menuZalogowany]
```

Na początek zwróć uwagę na sekcję `all`. Pojawiła się w niej dyrektywa `components`. W sekcji dla komponentów określiłeś, że komponent `menu_slotowe` jest wyłączony. W jaki sposób? Podałeś jako wartość nawiasy kwadratowe `[]`. Oznaczają one, że komponent slotowy ma być wyłączony.

W sekcjach konfigurujących widoki ponownie użyłeś dyrektywy `component`. Tutaj tak naprawdę uwidacznia się siła, która drzemie w połączeniu komponentu i slotu. Otóż dla widoku `profilSuccess` podałeś, że komponent `menu_slotowe` pochodzi z modułu `kompo` ↪`nent_slotowy` i jest nazwany `menuProfil`. Podobnie jak widok `zalogowanySuccess`, komponent `menu_slotowe` pochodzi z modułu `komponent_slotowy` i jest nazwany `menu` ↪`Zalogowany`. Łatwo zauważyć, że konfiguracja wymaga podania dwóch wartości: nazwy modułu, w którym zapisany jest komponent generujący kod, oraz nazwy komponentu. Kod HTML wygenerowany przez odpowiedni komponent wstawiany jest w layoucie w miejsce metody `include_component_slot('menu_slotowe')`.

Czas na testy. Wyczyść pamięć podręczną i wywołaj poniższe żądania:

http://127.0.0.1/nauka_symfony/web/widok.php/komponent_slotowy/zalogowany

http://127.0.0.1/nauka_symfony/web/widok.php/komponent_slotowy/profil

Jeżeli wszystko wykonałeś poprawnie, każde żądanie powinno mieć swoje własne menu. Na rysunku 4.4 możesz zobaczyć zrzut ekranowy menu dla akcji profil.

Rysunek 4.4.
Menu uwzględniające komponent menuProfil

Podsumowanie

Zaprezentowane w tym rozdziale przykłady nie wyczerpują zagadnień związanych z widokami Symfony. Stanowią jednak bardzo mocne podstawy i pozwalają na rozpoczęcie przygody z bardziej wymagającymi projektami.

O czym mogłeś przeczytać w tej części:

- pomocniki widoków,
- instalowanie i konfigurowanie layoutu,
- elementy widoku takie jak partiale, komponenty, komponenty slotowe,
- używanie slotów,
- konfiguracja przy użyciu *view.yml* oraz obiektu `Response`,
- zasoby zewnętrzne.