INSTYTUT INŻYNIERII I GOSPODARKI WODNEJ
POLITECHNIKA KRAKOWSKA im. TADEUSZA KOŚCIUSZKI

Gorzan Marcin

SYSTEM GROMADZENIA POMIARÓW HYDROMETEOROLOGICZNYCH WYKORZYSTUJĄCY FRAMEWORK SYMFONY

praca magisterska

studia dzienne

kierunek studiów: **informatyka**

specjalność: **informatyka stosowana w inżynierii środowiska**

promotor: **dr inż. Robert Szczepanek**

nr pracy: 2183

KRAKÓW 2008

ul. Warszawska 24, 31-155 Kraków tel/fax (+48 12) 628 20 41 e-mail: sekretariat@iigw.pl internet: www.iigw.pl

Składam serdeczne podziękowania

moim Rodzicom

za wsparcie i wiarę we mnie przez wszystkie lata edukacji,

Dr inż. Robertowi Szczepankowi

za cenne uwagi, poświęcony mi czas i pomoc udzieloną przy pisaniu niniejszej pracy.

1. *Wstęp* ... 4
   1.1. Cel Pracy ... 5
   1.2. Problematyka oraz zakres pracy ... 5
2. *Framework Symfony* ... 7
   2.1. Podstawowe zagadnienia dotyczące framework-ów ... 8
   2.2. Object Oriented Programming (OOP) w PHP ... 8
   2.3. RAD ... 10
   2.4. Ujęcie historyczne powstania framework-a ... 11
   2.5. Symfony implementacją wzorca MVC ... 13
   2.6. Budowa, struktura oraz organizacja aplikacji opartej o framework symfony 16
   2.7. Warstwa kontrolera ... 21
   2.8. Warstwa widoku ... 28
   2.9. Warstwa modelu ... 33
   2.10. Linki i system „Routingu” ... 37
   2.11. Generatory ... 41
   2.12. Testowanie ... 44
   2.13. Uruchomienie i konfiguracja projektu symfony ... 49
   2.14. Podsumowanie ... 51
3. *Wymagania projektowanego systemu* ... 53
   3.1. Wymagania funkcjonalne ... 54
      3.1.1. Podstawowe wymagania systemu ... 54
      3.1.2. Warstwa dodawania danych ... 54
      3.1.3. Warstwa edycji danych ... 55
      3.1.4. Wyszukiwanie oraz prezentacja danych ... 55
      3.1.5. Bezpieczeństwo ... 56
      3.1.6. Wygląd oraz interfejs ... 56
   3.2. Wymagania niefunkcjonalne ... 56
      3.2.1. Użytkownicy ... 56
      3.2.2. Obsługa błędów ... 56
      3.2.3. Wymagania sprzętowo systemowe ... 57
4. *Implementacja* ... 58
   4.1. Struktura bazy danych ... 59
   4.2. Opis modułów: ... 59
   4.3. Moduł czujniki. ... 60
   4.4. Moduł stacje. ... 62
   4.5. Moduł pomiary. ... 64
   4.6. Moduł import. ... 66
   4.7. Moduł wyszukiwania. ... 66
   4.8. Moduł security. ... 68
   4.9. Obsługa błędów. ... 76
5. *Podsumowanie* ... 77
6. *Wnioski* ... 80
7. *Bibliografia* ... 83
8. *Spis Listing'ów* ... 85
9. *Spis rysunków* ... 89

Usunięto (×23)

1. ***Wstęp***

1.1. Cel Pracy

Celem poniższej pracy jest stworzenie systemu gromadzenia danych pomiarowych z pomiarów hydrometeorologicznych. Systemu, którego zadaniem jest uprościć i przyspieszyć proces gromadzenia owych danych oraz w znaczny sposób ułatwić do nich dostęp oraz ich prezentacje. Aplikacja będzie pozwalała przechowywać w bazie dane pomiarowe z posterunków badawczych Politechniki Krakowskiej. Baza oraz sam system będzie przygotowany w ten sposób, aby mógł zawierać dane z pomiarów tradycyjnych oraz automatycznych. System ma usprawnić dostęp do danych pomiarowych na potrzeby różnorakich badań i obliczeń. Dzięki rozbudowanej warstwie prezentacji pozwoli na wyszukanie, wyświetlenie oraz eksport pomiarów do użytecznych formatów.

Drugim równie ważnym celem, jaki stawia sobie praca jest przedstawienie oraz przybliżenie nowoczesnej technologii tworzenia systemów internetowych. Pokazanie ewolucji środowiska PHP oraz udowodnienie że tworzenie kompletnych, wydajnych jak i bezpiecznych systemów internetowych wcale nie musi być żmudną pracą lecz interesująca przygodą. Przytoczone zostaną podstawowe mechanizmy projektowania oraz implementacji aplikacji opartej o szablon symfony oraz zostanie wspomniany szereg rozwiązań, które wprowadza wybrana technologia, a które w znacznym stopniu ułatwiają i przyspieszają produkcję zaawansowanych systemów internetowych.

1.2. Problematyka oraz zakres pracy

Gromadzenie pomiarów hydrometeorologicznych jest jednym z ważniejszych elementów działalności wielu jednostek związanych z ochroną środowiska, meteorologią, ochroną przeciwpowodziową jak i również instytucji naukowych. Pomiary takie są źródłem wielu informacji na temat zmieniających się warunków środowiskowych oraz są wielce pomocne w przewidywaniu wszelakich zagrożeń związanych np. z powodziami. Dużym problemem dla tych instytucji jest gromadzenie owych danych pomiarowych. Przeważnie trzymane są one w postaci plików kartek

papieru z zapisanymi pomiarami. Jeżeli weźmiemy pod uwagę ilość stacji, typ pomiarów oraz czas ich mierzenia możemy sobie wyobrazić jak dużo mogą one zajmować. Taki sposób ich przetrzymywania nastręcza wiele oczywistych problemów. Uporczywe przeszukiwanie stosów papierów nie okazuje się jedynym problemem. Często takie pomiary mogą zaginąć lub przypadkowo zostać zniszczone. Dlatego też poniższej pracy postawiono za cel stworzenie systemu komputerowego, który pomoże takie dane przechować, skatalogować i umożliwić do nich szybki i łatwy dostęp.

Początek pracy poświęcony zostanie podstawom teoretycznym tworzonego systemu. W rozdziale tym pokrótce opiszę wybraną technologię używaną do tworzenia aplikacji webowych a następnie przedstawię i uzasadnię swój wybór. Kolejnym krokiem będzie przedstawienie informacji z zakresu inżynierii oprogramowania. Opiszę konwencje programowania obiektowego na przykładzie wybranego języka. Przybliżę problematykę wzorców projektowych oraz pojęcia frameworka. Tutaj skoncentruje się na opisie wybranego przeze mnie frameworka symfony. Opiszę jego strukturę, budowę oraz możliwości. Przybliżę historię powstawania frameworka oraz przybliżę pojęcie RAD (Rapid Aplication Development). Podam kilka informacji na temat integracji symfony z bazami danych. Opisze dokładniej każdą z warstw systemu opartego o symfony. Zwrócę uwagę na możliwości symfony dotyczące automatycznego generowania dużych części aplikacji oraz dostępnych mechanizmów testowania. Głównym zadaniem kolejnego etapu będzie przedstawienie samego procesu budowy aplikacji począwszy od zebrania wymagań, stworzenia koncepcji i projektu systemu poprzez proces implementacji, oraz wdrażania. Opiszę dokładnie strukturę systemu, budowę oraz zadania poszczególnych modułów, które tworzyłem oraz przedstawię dokładniej bardziej interesujące rozwiązania, które zastosowałem.

2. ***Framework Symfony***

Aplikacja webowa jest to ogólna nazwa dla programu, który działa na maszynie podłączonej do sieci zwanej serwerem i komunikuje się z użytkownikiem za pomocą przeglądarki internetowej. Różni się ona od zwykłej statycznej strony WWW tym, że zakłada w swym działaniu interakcje z użytkownikiem. Działanie aplikacji webowych umożliwia serwer WWW, Np. Apache, IIS. Do przygotowania samej aplikacji używa się różnych mechanizmów (np.**CGI**, **ASP**) i języków (np. **PHP, JPS, Java ,C#**). Przykładami aplikacji webowej mogą być internetowe serwisy bankowe, Allegro czy też Nasza Klasa. (Wiki)

2.1. Podstawowe zagadnienia dotyczące framework-ów

Framework (rama projektowa, szkielet) to w programowaniu struktura wspomagająca tworzenie, rozwój i testowanie powstającej aplikacji. Z reguły na framework składają się programy wspomagające, biblioteki kodu źródłowego i inne podobne narzędzia.

To szkielet działania aplikacji, który zapewnia podstawowe mechanizmy i może być wypełniany właściwą treścią programu. Np. w programowaniu gier na szkielet może składać się utworzenie pustego okna, kod inicjalizacji i finalizacji biblioteki graficznej, a także dodatkowe moduły wspomagające, jak wczytywanie tekstur z różnych formatów plików, funkcje rysujące podstawowe figury geometryczne, tekst. W programowaniu sieciowym może to być zespół klas do obsługi połączeń z bazami danych, klas walidujących formularze, obsługujących sesje lub ogólnie konfigurujących całą aplikacje. Przykładami takich oto struktur mogą być: CakePHP, Smarty, Zend Framework, ale także platforma .NET, JSP, NetBeans. Wśród nich znajduje się wybrany przeze mnie framework symfony.

2.2. Object Oriented Programming (OOP) w PHP

Pierwsza wersja PHP, rozpowszechniana pod nazwą PHP/FI (Personal Home Page/Forms Interpreter), została stworzona przez duńskiego programistę Rasmusa Lerdorfa. W roku 1994 napisał on zestaw skryptów Perla służących do monitorowania internautów, którzy odwiedzali jego witrynę. Gdy ruch na stronach stał się duży, przepisał je w języku C, rozszerzając przy tym funkcjonalność samej aplikacji.

Niedługo później użytkownicy zaczęli prosić go o możliwość użycia tych narzędzi na swoich stronach, zatem 8 czerwca 1995 roku autor udostępnił ich kod źródłowy pod nazwą PHP Tools 1.0. W 1997 roku pojawiło się PHP/FI 2.0. Jedyne oficjalne wydanie nastąpiło w listopadzie 1997 roku. W czerwcu 1998 roku ogłoszono PHP 3.0 jako następcę PHP/FI, którego zaprzestano rozwijać. Zimą 1998 roku, krótko po wydaniu PHP 3.0, zaczęto przepisywać ponownie kod źródłowy PHP obierając za główne cele zwiększenie wydajności działania większych i bardziej złożonych aplikacji. Wersja 4.0 była gotowa w 2000 roku. W 2002 rozpoczęto prace nad dosyć znaczną modyfikacją języka. Aktualnym celem programistów stało się stworzenie w pełni obiektowego języka mogącego konkurować z pojawiającymi się na rynku coraz to nowocześniejszymi rozwiązaniami. W wersji 5.0 pojawił się zatem całkowicie nowy model programowania obiektowego, co spowodowało utratę pełnej kompatybilności z poprzednimi wersjami PHP. W ramach tego modelu zmieniony został sposób reprezentacji obiektów. W wersjach wcześniejszych obiekt był jednocześnie zmienną, co sprawiało duże trudności, dlatego też w wersji 5 (na wzór Javy) zmienna obiektowa stała się jedynie referencją do właściwego obiektu. Obecnie operacja przypisania powoduje powstanie drugiej referencji wskazującej na ten sam obiekt. Od 2005 roku trwają prace nad PHP6.0, obecnie ta wersja znajduje się w fazie programowania.

Idea przyświecająca programowaniu obiektowemu mówi, że aplikacja może być postrzegana jako ogólny zbiór części oraz obiektów połączonych wzajemnymi zależnościami i wpływającymi na siebie. Podejście to jest zgoła inne od proceduralnego, które mówi, że programy oraz aplikacje to zbiory funkcji bądź instrukcji dla komputera.

PHP5 implementuje większość paradygmatów OOP (Object Oriented Programming) takich jak:

- Klasy (implementacja obiektu , jego konkretyzacja), klasy abstrakcyjne, interfejsy, klasy i metody finalne

- Obiekty

- Dziedziczenie

- obsługa błędów try-catch

Podejście obiektowe w wersji 5 php pozwala nam na implementacje wzorców projektów dla programowania obiektowego. Podstawowym wzorcem projektowym, który stosowany jest w budowie framework-ów dla aplikacji webowych jest wzorzec MVC (Model View Control). Wzorzec ten jako główne założenie przyjmuje podział aplikacji na niezależne od siebie warstwy: modelu, widoku i kontroli.

2.3. RAD

Programowanie aplikacji internetowych było powolną pracą. Zazwyczaj typowe cykle tworzenia aplikacji (na przykład takie jak Rational Unified Process) nie mogły się rozpocząć bez zdefiniowania wymagań, narysowania wielu diagramów UML (Unified Modeling Language) i spisania dokumentacji wstępnej. Było to spowodowane ogólną szybkością ówczesnego tworzenia aplikacji, brakami we wszechstronności języków programowania. Należało zbudować aplikację, skompilować, uruchomić ponownie i kto wie co jeszcze zrobić zanim mieliśmy okazję ujrzeć ją działającą. No i przede wszystkim klienci cały czas zmieniali zdanie co do funkcjonalności systemu.

Dziś interesy prowadzone są dużo szybciej a klienci mają zwyczaj częstych zmian swojego zdania w samym środku procesu tworzenia aplikacji. Oczywiście zawsze liczą, że zespół programistów dostosuje się do nowych pomysłów i potrzeb oraz przebuduje aplikację jak najszybciej. Na szczęście wykorzystanie skryptowych języków programowania pozwala na swobodne wykorzystanie takich strategii programowania jak Rapid Application Development (RAD) lub Agile.

Jedną z idei stojących za wspomnianymi metodologiami jest rozpoczęcie tworzenia oprogramowania jak najszybciej to możliwe tak, aby klient miał wgląd w prototyp w chwili jego tworzenia oraz mógł oferować alternatywne kierunki rozwoju. Podejście takie pozwala na budowanie aplikacji w sposób iteracyjny. Oznacza to tworzenie kolejnych wersji bogatszych funkcjonalnie w stosunku do pierwowzoru w krótkich odstępach czasu.

Dla programisty konsekwencje są liczne. Nie musi on myśleć o przyszłości, kiedy implementuje daną funkcjonalność. Metoda implementacji powinna być na tyle prosta

na ile jest to możliwe. Zasadę tą świetnie ilustruje zasada KISS, czyli skrót od maksymy Keep It Simple, Stupid.[1]

Kiedy wymagania się zmieniają lub wymagana jest dodatkowa funkcjonalność część kodu zazwyczaj musi być napisana na nowo. Proces ten zwany jest refaktoringiem i zdarza się dość często w przypadku tworzenia aplikacji internetowych. Jedne fragmenty kodu są przenoszone w inne miejsca zgodnie ze swoją naturą (funkcje używane w rożnych modułach do bibliotek globalnych) oraz przeznaczeniem inne natomiast są zamieniane na mniejsze fragmenty zgodnie z zasadą Don't Repeat Yourself (DRY) co po polsku oznacza po prostu: nigdy nie powtarzaj sam siebie.

Aby mieć pewność, że aplikacja działa pomimo dokonywania w niej stałych zmian wymagany jest cały zestaw testów jednostkowych, które można wykonywać automatycznie. Dobrze napisane testy jednostkowe mogą być solidnym sposobem, aby upewnić się, że wszystko działa poprawnie oraz nie ma żadnych problemów spowodowanych przez refaktoryzację kodu. Pewne metodologie przewidują nawet pisanie testów kodu zanim ów kod powstanie. Działanie takie jest zwane test-driven development (TDD).

2.4. Ujęcie historyczne powstania framework-a

Pierwsza wersja frameworka symfony została opublikowana w październiku 2005 r. przez jego fundatora Fabiena Potenciera. Osoba ta jest właścicielem firmy Sensio (http://www.sensio.com) zajmującej się marketingiem. Autor poświęcił dużo czasu poznając i studiując narzędzia do tworzenia aplikacji w php. Nie zdołał on znaleźć odpowiednich dla siebie, które spełniłoby w pełni jego oczekiwania. Kiedy zostało wydane PHP5 twórca stwierdził, że w pełni funkcjonalny i obiektowy język może posłużyć do zbudowania dobrego frameworka. Fabien pracował dłuższy czas nad strukturą framworka. Inspirował się takimi produktami jak Mojavi czy Rubi on Rails. Swój framework oparł o model MVC.

Początkowo Fabien stworzył symfony dla projektów realizowanych przez Sensio, ponieważ efektywny framework był doskonałym narzędziem do szybszego oraz

[1] „*The Definitive Guide to Symfony*”*- Introducing Symfony - tłumaczenie.*

bardziej wydajnego tworzenia aplikacji. Sprawiło to, że tworzenie aplikacji internetowych stało się bardziej intuicyjne a w rezultacie aplikacje stały się bardziej solidne i prostsze w rozwoju. Pierwszym poważnym testem dla symfony był serwis typu e-commerce sprzedający odzież. Później symfony zostało wykorzystane również do budowy kilku mniejszych projektów. [2]

Po pomyślnych wdrożeniach w kolejnych projektach Fabien postanowił opublikować symfony na licencji Open Source.

Symfony to jeden z bardziej rozbudowanych i przemyślanych framework-ów PHP5. Jego przejrzysta struktura, czytelny kod, dobre praktyki programistyczne i aktywne środowisko deweloperów pozwalają przyspieszyć produkcję aplikacji internetowych. Twórcy systemu wykorzystali przy jego tworzeniu wiele już dostępnych rozwiązań wychodząc z założenia, że po co na nowo wymyślać coś co już jest i całkiem dobrze działa. Takimi rozwiązaniami są: Propel, Prototype, Prado. [3]

Propel jest to typowy ORM (Object-Relational Mapping), czyli narzędzie tworzące model obiektowo relacyjny dla bazy danych. Propel tworzy zestaw klas do manipulowania bazą danych. [4]

Prototype jest biblioteką JavaScript, zawierającą funkcję odpowiedzialne za żądania AJAX-owe.

Prado jest również frameworkiem opartym o wzorzec MVC, z którego symfony czerpie ogólne „know how".

---

[2] „T*he Definitive Guide to Symfony*"- *Introducing Symfony - tłumaczenie*
[3] "PHP solution" nr 27 artykuł – "Symfony Web Aplication ramework" – Jakub Zalas
[4] „T*he Definitive Guide to Symfony*"- *Introducing Symfony- tłumaczenie*

2.5. Symfony implementacją wzorca MVC

MVC (*Model-View-Controller*) - *Model-Widok-Kontroler* to wzorzec projektowy w informatyce, którego głównym założeniem jest wyodrębnienie trzech podstawowych komponentów aplikacji:

- modelu danych
- interfejsu użytkownika
- logiki sterowania

w taki sposób, aby modyfikacje jednego komponentu minimalnie wpływały na pozostałe. Czasem, w odniesieniu do MVC stosuje się nazwę "modelu trójwarstwowego". MVC powstał w 1978 roku.

By pokazać w jaki sposób przebiega implementacja wzorca MVC stworzyłem krótki program, który łączy się z bazą danych, wybiera odpowiednie rekordy oraz prezentuje je w przeglądarce. Dla porównania stworzyłem również standardowy program (płaska aplikacja) nie korzystający ze wzorca spełniający taką sama funkcjonalność.

Listing 2.5 Płaska aplikacja

```
<?php
// Connecting, selecting database
$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('baza', $link);

// Performing SQL query
$result = mysql_query('SELECT date, title FROM post', $link);
?>
<html>
  <head>
    <title>Lista Postów</title>
  </head>
  <body>
   <h1>Lista</h1>
   <table>
     <tr><th>Data</th><th>Tytuł</th></tr>
<?php
// Wyświetlenie wyniu w HTML
while ($row = mysql_fetch_array($result, MYSQL_ASSOC))
{
echo "\t<tr>\n";
printf("\t\t<td> %s </td>\n", $row['date']);
printf("\t\t<td> %s </td>\n", $row['title']);
echo "\t</tr>\n";
```

```
}
?>
    </table>
  </body>
</html>

<?php

mysql_close($link);
?>
```

Lista zalet takiego sposobu programowania kończy się na tym, że jest to szybkie do napisania. Oto największe wady takiego rozwiązania.

- Nie ma obsługi błędów przy połączeniu z bazą danych
- Kod php jest pomieszany z html-em co zmniejsza znacznie jego przejrzystość
- Kod uzależniony jest od jednego serwera bazy danych

Rozdzielmy więc ten kod na warstwy w myśl wzorca MVC. Pierwsza rzeczą, którą zrobimy będzie uniezależnienie naszego programu od bazy danych. Umieścimy w pliku model.php obsługę połączenia z bazą wraz z funkcją, która zwróci nam żądane dane. :

Listing 2.6 model.php – fragment kodu prezentujący zawartość pliku model

```
<?php

function getAllPosts()
{
  // Connecting, selecting database
  $link = mysql_connect('localhost', 'myuser', 'mypassword');
  mysql_select_db('baza', $link);

  // Performing SQL query
  $result = mysql_query('SELECT date, title FROM post', $link);

  // Filling up the array
  $posts = array();
  while ($row = mysql_fetch_array($result, MYSQL_ASSOC))
  {
     $posts[] = $row;
  }





  // Closing connection
  mysql_close($link);

  return $posts;
}
```

```
?>
```

Teraz gdy będziemy chcieli zmienić silnik bazodanowy, zmiany dokonywane będą tylko w tym pliku natomiast cała reszta pozostanie bez zmian.

Kolejnym krokiem będzie rozdzielenie logiki biznesowej od prezentacji. Kod html oraz php w jednym pliku nie wygląda przejrzyście więc rozdzielamy go na dwa pliki:

Listing 2.7 Kontroler.php – fragment kodu kontrolera

```
<?php

// Requiring the model
require_once('model.php');

// Retrieving the list of posts
$posts = getAllPosts();

// Requiring the view
require('view.php');
```

Listing 2.8 viev.php – kod przedstawia widok

```
<html>
  <head>
    <title>Lista postów</title>
  </head>
  <body>
    <h1>Lista</h1>
    <table>
      <tr><th>Data</th><th>Tytuł</th></tr>
    <?php foreach ($posts as $post): ?>
      <tr>
        <td><?php echo $post['date'] ?></td>
        <td><?php echo $post['title'] ?></td>
      </tr>
    <?php endforeach; ?>
    </table>
  </body>
</html>
```

Jak widać dzięki takiemu rozgrupowaniu kodu oprócz przejrzystości zyskujemy powtarzalność. Możemy wykorzystać kod w innym projekcie zmieniając jedynie

warstwę widoku lub zmieniając serwer bazy danych bez żadnego konfliktu z innymi częściami aplikacji. [5]

2.6. Budowa, struktura oraz organizacja aplikacji opartej o framework symfony

Implementacja wzorca MVC w symfony opiera się o skrypty przedstawione poniżej:

- Warstwa modelu (Model layer)
  - Zespół klas opisujących model (Database abstraction)
  - Zespół funkcji odpowiedzialnych za dostęp do danych (Data access)
- Warstwa widoku (View layer)
  - Widok (View)
  - Szablon (Template)
  - Layout
- Warstwa kontroli (Controller layer)
  - Kontroler główny (Front controller)
  - Akcje (Action)

Wydaje się iż 7 skryptów na obsłużenie jednej strony to trochę dużo, ale przyjrzyjmy się problemowi bliżej. Na początek trzeba wspomnieć o tym, iż warstwa front kontroler oraz Layout są wspólne dla całej aplikacji. Można mieć ich oczywiście kilka, ale w danym momencie wymagana jest tylko jedna z nich. Element Front kontroler jest czysto logicznym komponentem MVC, w którym nie piszemy żadnego kodu. Symfony generuje go za nas. Identyczna sprawa jest z warstwą modelu, która jest generowana na podstawie danych dostarczonych przez programistę w plikach konfiguracyjnych. To samo tyczy się logiki wyświetlania View Logic (więcej informacji w rozdziale 2.8, 2.9). Ostatecznie dla programisty zostaje tak naprawdę kodowanie 3 elementów: layout, template, oraz action.

[5] Fragmenty kodów z rozdziału „Expolring Symfony Code" – „T*he Definitive Guide to Symfony*"

Implementacja MVC Symfony udostępnia nam wiele klas do obsługi różnorakich zdarzeń podczas działania aplikacji. Podstawowe z nich to:

- `sfController` – dekoduje żądania i wysyła je do akcji

- `sfRequest` – przechowuje elementy typu request (parametry, ciasteczka, nagłówki)

- `sfResponse` – zawiera elementy response, czyli elementy które zostaną dekodowane do Html'a i wysłane do użytkownika

- `sfContext` – przechowuje referencje do wszystkich podstawowych obiektów i danej konfiguracji systemu.

Znając już mniej więcej budowę rożnych komponentów symfony czas przybliżyć strukturę w jakiej są one zorganizowane. Struktura ta opiera się o projekt, który z kolei ma budowę drzewa. Projekt jest zbiorem serwisów oraz funkcji ,które działają pod jednym adresem domenowym, dzieląc ze sobą warstwę modelu. Aplikacje są niezależnymi od siebie elementami, które mogą działać odseparowane. Doskonałym przykładem może być system CMS (Content Management System) służący do zarządzania treścią strony, gdzie jedną aplikacją jest tak zwany frontendem (przód), czyli to co widzi użytkownik oglądający stronę internetową. Druga aplikacji to backend (tył), czyli część administracyjna, w której redaktorzy zmieniają zawartość stron, nowości czy też tworzą nowe. Docelowo projekt może zawierać kilkanaście mini-aplikacji, w tym przypadku serwisów internetowych, po których można się swobodnie poruszać dzięki hiperłączom zawartym w stronach.

Każda aplikacja składa się z jednego lub wielu modułów. Moduł zwykle reprezentuje stronę lub grupę stron, które spełniają pewną funkcjonalność. W przypadku naszej aplikacji przykładem może być moduł dodawania pomiarów czy też moduł tworzenia stacji pomiarowych. Moduły przechowują akcje, które reprezentują wszystkie zadania dla danego modułu. Przykładem może być akcja generowania formularza do dodawania pomiaru czy też zapisu danych do bazy. Pozostałymi elementami projektu są:

- Baza Danych Np MySql czy PostgreSql.
- Pliki statyczne (HTML, obrazki, JavaScript , pliki stylów)
- Pliki wysyłane do serwera przez użytkowników lub administratorów
- Klasy i biblioteki PHP
- Inne biblioteki
- Pliki ”batch” (skrypty uruchamiane z linii komend lub przez cron)
- Pliki Logów (logowanie działania aplikacji)
- Pliki konfiguracyjne

Listing 2.9 Struktura projektu stworzonego w symfony

```
apps/
  frontend/
  backend/
cache/
config/
data/
  sql/
doc/
lib/
  model/
log/
plugins/
test/
  bootstrap/
  unit/
  functional/
web/
  css/
  images/
  js/
  uploads/
```

`apps/` - Zawiera po jednym katalogu dla każdej aplikacji w danym projekcie.

`cache/` Zawiera “zachowaną” konfiguracje dla projektu oraz (po włączeniu) tożsamą wersje akcji i szablonów dla poszczególnej aplikacji. Używany jest, by przyspieszyć działanie aplikacji.

`config/` Przechowuje ogólną konfiguracje projektu.

`data/ -` Tu przechowywane są pliki takie jak schematy bazy danych, pliki SQL oraz bazy SQLight.

`doc/` Katalog przygotowany do przechowywania dokumentacji projektu.

`lib/` Katalog dedykowany dla klas lub bibliotek zewnętrznych. Będą dostępne dla wszystkich aplikacji.

`model/` Katalog zawiera klasy warstwy modelu.

`log/` Zawiera pliki logów generowane przez symfony. Może również zawierać logi serwera webowego logi bazy danych lub logi dotyczące wszystkiego co związane z projektem.

`plugins/` Katalog przechowuje wtyczki (pluginy) w pewnym stopniu ułatwiające tworzenie różnych elementów aplikacji. Najczęściej są tworzone przez społeczność korzystającą z frameworka.

`test/` Zawiera testy jednostkowe oraz funkcjonalne. Symfony podczas tworzenia projektu sam dodaje kilka podstawowych testów.

`web/` Katalog główny dla serwera webowego. Jedyne pliki jakie są widoczne z Internetu. Na ten katalog wskazuje wirtualny host lub link symboliczny w serwerze webowym.

Listing 2.10 Struktura pojedynczej aplikacji

```
apps/
  [application name]/
    config/
    i18n/
    lib/
    modules/
    templates/
      layout.php
```

`config/` Zawiera pliki konfiguracyjne dla danej aplikacji

`i18n/` Zawiera pliki lokalizacyjne dla aplikacji. Przeważnie są to pliki z tłumaczeniami dla interfejsu.

`lib/` Klasy i biblioteki widziane tylko w danej aplikacji

`modules/` Wszystkie moduły danej aplikacji.

`templates/` – Zawiera globalny szablon dla wyglądu strony, który będzie dzielony dla wszystkich modułów.

Listing 2.11 Struktura modułu

```
apps/
  [application name]/
    modules/
      [module name]/
          actions/
            actions.class.php
          config/
          lib/
          templates/
            indexSuccess.php
          validate/
```

Action / Zawiera właściwie jeden plik klasę zwany action.calss.php, w którym znajdują się wszystkie akcje dla danego modułu. Przy tworzeniu aplikacji znajduje się tam domyślnie akcja indexExecute().

Config/ Konfiguracja danego modułu.

Lib/ Klasy i biblioteki dla danego modułu.

Temlates/ zawiera szablony dla każdej z akcji modułu. Standardowo przy tworzeniu aplikacji symfony znajduje się tu szablon o nazwie indexSucces.php, który odpowiada akcji indexExecute().

Validate/ Zawiera pliki konfiguracyjne do obsługi walidacji danego modułu[6].

[6] Struktura projektu i opis fragmentów – „Expolring Symfony Code" – „T*he Definitive Guide to Symfony*"

2.7. Warstwa kontrolera

W symfony warstwa kontrolera jest odpowiedzialna za łączenie ze sobą logiki baz danych i prezentacji. W tej warstwie zachodzą wszystkie operacje na danych zaciągniętych do bazy, ładowanie konfiguracji, oraz zarządza wyświetlaniem akcji. Tak podstawowe zadania warstwy kontrolera przedstawia nam autor książki „T*he Definitive Guide to Symfony*":

- Akcje zawierają logikę aplikacji. Sprawdzają spójność żądania i przygotowują dane potrzebne warstwie prezentacji.
- Obiekty żądania, odpowiedzi oraz sesji udostępniają parametry żądania, nagłówki odpowiedzi i trwałe dane użytkownika. Są intensywnie wykorzystywane w warstwie kontrolera.
- Filtry są fragmentami kodu wykonywanymi przy każdym żądaniu, oraz po wykonaniu akcji. Przykładem są filtry bezpieczeństwa oraz walidacji, powszechnie używane w aplikacjach web. Możesz rozszerzyć funkcjonalność frameworka poprzez stworzenie własnych filtrów.

Front Controller

Wszystkie żądania są obsługiwane przez pojedynczy Front Controller, który jest punktem wejścia do całej aplikacji w danym środowisku. Jest on zawarty w pliku index.php w katalogu dostępnym dla serwera WWW.

Front Controller po otrzymaniu żądania wyświetlenia jakiejś akcji tłumaczy adres żądania za pomocą systemu routingu. Po dopasowaniu przetłumaczonego URL-a do odpowiedniej akcji wywołuje ją. Przykładowo żądanie poniżej spowoduje, że Front Controller index.php wyśle żądanie do akcji myAction w module mymodule:

```
http://localhost/index.php/mymodule/myAction
```

Można powiedzieć ze Front Controller dokonuje rozdysponowania żądań do akcji. Dodatkowo, wykonuje on kod wspólny dla wszystkich akcji, wliczając w to takie elementy działania aplikacji opartej o symfony jak:

1. Definiowanie stałych jądra.
2. Odnajdywanie bibliotek symfony.
3. Ładowanie i inicjacja klas jądra.
4. Ładowanie konfiguracji.
5. Dekodowanie URL żądania, w celu odczytania nazwy akcji do wykonania i parametrów żądania.
6. Jeżeli akcja nie istnieje, to wykonanie przekierowania do akcji błędu 404.
7. Aktywacja filtrów (na przykład jeżeli żądanie wymaga sprawdzenia tożsamości).
8. Wywołanie filtrów, pierwsze przejście.
9. Wywołanie akcji i wygenerowanie widoku.
10. Wywołanie filtrów, drugie przejście.
11. Zwrócenie odpowiedzi. [7]

Typowy Front Controller wygląda jak na listingu 2.12

Listing 2.12 Typowy Front Controller utworzony przez symfony [8]

```
define('SF_ROOT_DIR',    realpath(dirname(__FILE__).'/..'));
define('SF_APP',         'myapp');
define('SF_ENVIRONMENT', 'prod');
define('SF_DEBUG',       false);

require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATO
R.SF_APP.DIRECTORY_SEPARATOR.'config'.DIRECTORY_SEPARATOR.'config.php'
);

sfContext::getInstance()->getController()->dispatch();
```

Akcje

Akcje są reprezentowane prze grupy metod zawarte w klasie moduleNameActions dziedziczącej z klasy sfActions. Każda z metod swą nazwę rozpoczyna od przedrostka execute a następnie występuje nazwa akcji np.: executeIndex() Klasa akcji danego

[7], [8] „Inside The Controller Layer” – „T*he Definitive Guide to Symfony*”

modułu jest zapisana w pliku actions.class.php w folderze actions/. Przykładowy wygląd pliku action.class.php

Listing 2.13 plik action.calss.php

```
class mymoduleActions extends sfActions
{
  public function executeIndex()
  {

  }
}
```

W celu wysłania żądania wykonania akcji, należy wywołać skrypt Front Controllera wraz z nazwą modułu i akcji podanymi jako parametry. Domyślną implementacją jest dodanie pary module_name/action_name do skryptu. Znaczy to, że akcja może zostać wywołana z następującym URL:

Listing 2.14 Wywołanie akcji w module

```
http://localhost/index.php/mymodule/index
```

Dodawanie kolejnych akcji wiąże się jedynie z dopisywaniem kolejnych metod typu execute (jak na listingu poniżej)[9].

Listing 2.15 Kilka akcji w action.class.php

```
class mymoduleActions extends sfActions
{
  public function executeIndex()
  {
    ...
  }
```

[9] Przykłady „Inside The Controller Layer” – „T*he Definitive Guide to Symfony*”

```
  public function executeList()
  {
    ...
  }
}
```

Pobieranie informacji wewnątrz akcji

Na listingu 2.16 pokazano możliwości pobierania informacji o kontrolerze w funkcji odpowiedzialnej za dana akcje. Metody klasy sfAction dają również dostęp do obiektów jądra symfony :

Listing 2.16 Przykładowa akcja[10]

```
class mymoduleActions extends sfActions
{
  public function executeIndex()
  {
    // Pobieranie parametrów żądania
    $password    = $this->getRequestParameter('password');

    // Pobieranie informacji o kontrolerze
    $moduleName  = $this->getModuleName();
    $actionName  = $this->getActionName();

    // Pobieranie obiektów jądra frameworka
    $request     = $this->getRequest();
    $userSession = $this->getUser();
    $response    = $this->getResponse();
    $controller  = $this->getController();
    $context     = $this->getContext();

    // Nadawanie wartości zmiennym akcji w celu przesłania ich do
szablonu
    $this->setVar('foo', 'bar');
    $this->foo = 'bar';            // Wersja skrócona
  }
}
```

Listing 2.17 Przechwytywanie danych z formularza

[10] „Inside The Controller Layer" – „T*he Definitive Guide to Symfony*"

```
$password     = $this->getRequestParameter('password');
```

Ten fragment kodu (*listing 2.17*) obrazuje dosyć popularną i często wykorzystywaną metodę przechwytywania informacji podczas przenoszenia pomiędzy akcjami. Za pomocą funkcji getRequestParametr() przechwytujemy wartości, które przesyłane są do danej akcji za pomocą metody POST lub GET, przy czym nie ma znaczenia dla funkcji jaka to metoda. Ważne są tylko nazwy parametrów. W tym przypadku będzie to wartość textboxa o id password przesłana metodą POST. Równie dobrze mogłaby być ona przesłana metodą GET (w linku) w następujący sposób:

http://localhost/index.php/mymodule/index?password=asdfg

Listnig 2.18 Wysyłanie do widoku

```
    $this->foo = 'bar' , $this->zmienna = $zmienna;
```

Taka akcja (*listing 2.18*) nazywa się popularnie wysłaniem do widoku. Będzie ona wtedy widoczna w warstwie widoku jako normalna zmienna ($foo, $zmienna) lub tablica zmiennych, której atrybuty lub wartości można tam wyświetlać. W ten sposób wysyłamy wyniki pewnych obliczeń wartości zwracane przez inne funkcje lub obiekty przetrzymujące rekordy zwrócone przez bazę danych.

Każdą akcję możemy zakończyć w różny sposób. Wartość zwracana przez metodę odpowiedzialną za akcje definiuje nam ostatecznie wygląd i zachowanie widoku. Do określenia parametrów widoku, symfony dostarcza nam gotowa klasę sfView. Stałe tej klasy służą do wyznaczania jaki szablon będzie załadowany. Zwykle każdą akcję kończy się w taki sposób jak na listingu 2.19. Jest to domyślne zakończenie akcji więc możemy je pominąć pisząc kod. Takie wywołanie spowoduje wyświetlenie szablonu przypisanego do danej akcji.

Listing 2.19 Zakończenie akcji

```
[php]
return sfView::SUCCESS;
```

Symfony wyszuka szablon o nazwie actionNameSuccess.php. Jeżeli konieczne jest wywołanie widoku błędu, to akcja powinna kończyć się następująco:

Listing 2.20 Zwrócenie widoku błędu

```
[php]
return sfView::ERROR;
```

Symfony wtedy odszuka plik szablonu o nazwie actionNameError.php.

Aby wywołać własny widok, użyjemy zakończenia: (*listing 2.21*)

Listing 2.21 Zwracanie własnego widoku

```
[php]
return 'MyResult';
```

Wtedy symfony poszuka szablonu actionNameMyResult.php.

Jeżeli chcemy by akcja nie wywoływała żadnego widoku (jest to wykorzystywane przy procesach wsadowych) stosujemy zakończenie z listingu 2.22:

Listing 2.22 Zakończenie bez widoku

```
[php]
return sfView::NONE;
```

Każdą z akcji można również zakończyć wywołaniem innej akcji. Na przykład akcja obsługująca wysyłanie formularzy metodą POST zazwyczaj przekierowuje do kolejnej akcji po zaktualizowaniu bazy danych. Klasa sfAction udostępnia dwie metody służące do tego typu działań:

- Przekierowanie (forwardowanie) wywołania do innej akcji *(listing 2.23)*

Listing 2.23 Sposób wymuszenia przejścia do innej akcji

```
[php] $this->forward('otherModule', 'index');
```

- Przekierowanie poprzez żądanie *(listing 2.24)*

Listing 2.24

```
[php] $this->redirect('otherModule/index');
$this>redirect('http://www.google.com/');
```

Symfony udostępnia również specjalny bardzo wygodny system przekierowań „zależnych". Metody forward404(), forward404If(), forward404() przyjmują one żądania albo zmienne jako argumenty i sprawdzając ich poprawność wysyłają na stronę błędu lub nie.

Listing 2.25 Przekierowanie na stronę błędu;

```
[php]
public function executeShow()
{
  $article = ArticlePeer::retrieveByPK($this-
>getRequestParameter('id'));
  if (!$article)
  {
    $this->forward404();
  }
$this->forward404If(!$article);
$this->forward404Unless($article);


}
```

Sesja

Obiekt sesji użytkownika jest dostępny w akcji poprzez wywołanie metody getUser() i jest instancją klasy sfUser. Klasa ta pozwala nam na przechowywanie dowolnego atrybutu (łańcuchy znaków, tablice, w tym asocjacyjne). Dane te dostępne są w całej aplikacji i w każdym module dopóki sesja nie wygaśnie lub zostanie zakończona

Listing 2.26 Przechowywanie atrybutów w sesji

```
[php]
class mymoduleActions extends sfActions
{
  public function executeFirstPage()
  {
    $nickname = $this->getRequestParameter('nickname');

    // Zapisz dane w sesji użytkownika
    $this->getUser()->setAttribute('nickname', $nickname);
```

```
  }

  public function executeSecondPage()
  {
    // Pobierz dane z sesji użytkownika, używając domyślnej wartości
    $nickname = $this->getUser()->getAttribute('nickname', 'Anonymous
Coward');
  }
}
```

Zakończenie sesji odbywa się przez usuniecie atrybutów użytkownika:

Listing 2.27 Zakończenie sesji.

```
$this->getUser()->getAttributeHolder()->clear();
```

Warstwa kontrolera ogólnie służy nam do budowania całej logiki aplikacji, obróbki danych uzyskanych z warstwy modelu i przesyłanie wyników do widoku. W tej warstwie zarządzamy sesją oraz prawami użytkownika. Tutaj prowadzona jest również walidacja oraz filtrowanie informacji przekazywanych podczas pracy z aplikacją.

2.8. Warstwa widoku

Widok jest odpowiedzialny za prezentowanie danych zwracanych przez poszczególne akcje. W symfony, widok składa się z oddzielnych części, z których każda jest zaprojektowana tak, aby mogła być łatwo modyfikowana przez osoby zajmujące się wyglądem witryn internetowych.

- Webdesignerzy zazwyczaj pracują nad szablonami (prezentacją danych pochodzących z aktualnej akcji), a także nad wyglądem (layoutem zawierającym kod jednakowy dla wszystkich stron). Te pliki są napisane w języku HTML z niewielką domieszką kodu PHP, który najczęściej uruchamia pomocniki (helper).
- W celu umożliwienia wielokrotnego wykorzystania konkretnych elementów, deweloperzy najczęściej „wkładają” fragmenty kodu do komponentów. Używają slotów i komponentów, aby mieć wpływ na więcej niż jeden obszar layoutu. Webdesignerzy mogą pracować na fragmentach szablonów równie wydajnie.
- Deweloperzy zwracają uwagę na YAML (uniwersalny język formalny przeznaczony do reprezentowania różnych danych w ustrukturalizowany

sposób) w plikach konfiguracyjnych (ustawienia odpowiedzi - response i innych elementów interfejsu) oraz na obiekt response. Podczas zawierania zmiennych w szablonach trzeba pominąć ryzyko ataku typu cross-site scripting, gdyż zasięg technik filtracyjnych obejmuje całkowicie dane pochodzące od użytkownika. [11]

Jednym z ciekawszych elementów, które udostępnią nam framework symfony podczas budowy widoku są pomocniki. Jest to zestaw funkcji PHP, które w pewnym stopniu przyspieszają tworzenie różnych elementów języka HTML podczas budowania strony. Zwracają one kod HTML niektórych znaczników HTML pozwalając na nieco szybsze budowanie formularzy lub komunikacji miedzy stronami. Przykładem może być pomocnik dla typowego okienka tekstowego.

Listing 2.28 Pomocnik (helper) input_tag()

```
<?php
echo input_tag('imie') ?>  co w powoduje wyświetlenie czegoś takigeo
=>
<input type="text" name="imie" id="imie" value="" />
```

Jak widać na przykładzie ilość znaków na przedstawienie tej samej funkcjonalności zmniejszyła się co najmniej dwukrotnie.

Pliki symfony zawierające definicje pomocników nie są ładowane automatycznie (dopóki zawierają funkcje, a nie klasy). Pomocniki są grupowane wg swoich zadań. Na przykład, wszystkie pomocniki zajmujące się wykonywaniem operacji na tekście są zdefiniowane w pliku, który nazywa się *TextHelper.php,* z którego z kolei są wywoływane pomocniki z grupy pomocników tekstu. Jeśli zachodzi potrzeba użycia pomocnika w szablonie, trzeba wcześniej załadować powiązaną z nim grupę pomocników deklarując ją za pomocą funkcji *use_helper().*

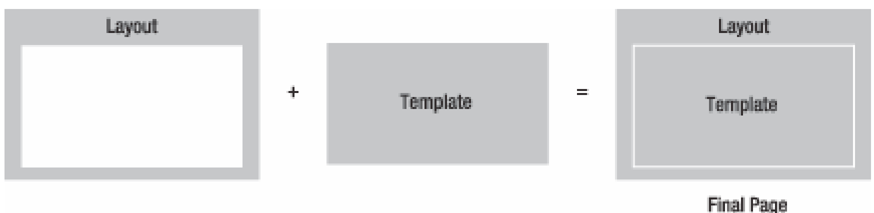
Poniższe helpery są ładowane standardowo w każdym szablonie i nie ma potrzeby deklarowania ich.:

[11] „Inside The Viev Layer" – „T*he Definitive Guide to Symfony*"

Helper: Wymagany do włączania pomocników (funkcja *use_helper()* w zasadzie także jest pomocnikiem)

- *Tag*: Podstawowy pomocnik etykiet, używany prawie przez wszystkie pomocniki
- *Url*: Pomocniki umożliwiające zarządzanie adresami URL oraz linkami
- *Asset*: Pomocniki rozpowszechnione w sekcji w kodzie HTML oraz dostarczające łatwe odwołania do zewnętrznych źródeł danych (obrazków, skryptów Javascript oraz arkuszy stylów)
- *Partial*: Pomocniki pozwalające na włączanie fragmentów szablonu
- *Cache*: Manipulacja cache-owanymi fragmentami kodu
- *Form*: Pomocniki umożliwiające tworzenie formularzy [12]

Większość pracy z widokiem opiera się na edycji i tworzeniu szablonów dla poszczególnych akcji. Szablon (ang. Template) jest nie pełnym dokumentem XHTML, w którym brakuje głównych znaczników <html>, <head> i <body>. Dzieje się tak dlatego, iż te znaczniki trzymane są w globalnym szablonie zwanym layout.php. Podczas ładowania strony szablon naszego modułu jest integrowany z layoutem aplikacji i wyświetlany jako całość. W pliku layout jest krótka funkcja PHP, która ładuje odpowiedni szablon, dla aktualnie wywołanego modułu i akcji. Całą sytuacje obrazuje poniższy schemat:



Rys. 2.1 Schemat ładowania wyglądu modułu do głównego layaut'u).
[źródło: opracowanie własne]

Przykładowy plik layoutu możemy zobaczyć poniżej. Zawiera on definicje wszystkich znaczników głównych, pomocników dotyczących meta Tagów, które ładują

[12] Lista dostępnych pomocników -„Inside The View Layer" – „T*he Definitive Guide to Symfony*"

odpowiednie informację zawarte w konfiguracji danej aplikacji oraz funkcje ładujące szablony.

Listing 2.29 Domyślny wygląd pliku layout.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <?php echo include_http_metas() ?>
  <?php echo include_metas() ?>
  <?php echo include_title() ?>
  <link rel="shortcut icon" href="/favicon.ico" />
</head>
<body>

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

</body>
</html>
```

Partials

Podczas budowy szablonów zdarza się, że często powtarzamy jakąś sekwencje kodu, by na przykład wyświetlić kilka informacji o podobnej budowie. Przykładem może być podobny wygląd okienek z wyświetlanym newsem. Taki fragment kodu doskonale nadaje się, by umieścić go w tzw. partialu, czyli powtarzalnym fragmencie kodu. Tak jak szablony, partiale lokalizuje się w katalogu templates/ i zawierają one przeważnie kod html z osadzonym php. Pliki typu partial zawsze zaczynają się od znaku _ przykładowo _mojpartial.php. W szablonie, partiali można używać niezależnie w jakim module one zostały zaimplementowane. Są one dostępne dla całej aplikacji. Takie zachowanie partiali jest bardzo wygodne choć powoduje mały problem. Partial nie ma dostępu do zmiennych danego modułu, ponieważ jest elementem globalnym. Można to szybko ominąć przesyłając do partiala zmienne w następujący sposób:

Listing 2.30 Ładowanie partiala

```
<p>Ciało szblonu </p>
<?php include_partial('mojpartial',
array('mojaZmienna' => $zmienna)
) ?>
```

Components

Czasem może zdarzyć się sytuacja, w której chcemy, aby nasz odseparowany fragment kodu wykonywał trochę bardziej skomplikowane zadania. Przeprowadzał pewne operacje na zmiennych lub odwoływał się do bazy danych. Według wzorca MVC takie operacje należało, by podzielić na warstwę logiczną i prezentacyjną. Tu z pomocą przychodzą nam komponenty (ang. Components), które są tak naprawdę partialami wzbogaconymi o warstwę logiczną. Tworząc komponenty najlepiej zbudować osobny moduł dla nich choć nie jest to konieczne. W module umieszczamy plik akcji `action/components.class.php,` który dziedziczy z klasy sfComponent, a w którym definiujemy akcje dla komponentu. Prezentacją jest natomiast partial o odpowiedniej do akcji nazwie. Budowa partiali jest bardzo podobna do budowy modułu.

Convention	Actions	Components
Logic file	actions.class.php	components.class.php
Logic class extends	sfActions	sfComponents
Method naming	executeMyAction()	executeMyComponent()
Presentation file naming	myActionSuccess.php	_myComponent.php

Rys 2.2 Rozróżnienie akcji dla modułu i komponentu.

Listing 2.31 Przykładowa klasa komponentu:

```
class newsComponents extends sfComponents
{
  public function executeHeadlines()
  {
    $c = new Criteria();
    $c->addDescendingOrderByColumn(NewsPeer::PUBLISHED_AT);
    $c->setLimit(5);
    $this->news = NewsPeer::doSelect($c);
  }
}
```

Oraz widok dla niego:

Listing 2.32 Widok dla komponentu

```
<div>
  <h1>Latest news</h1>
  <ul>
  <?php foreach($news as $headline): ?>
    <li>
      <?php echo $headline->getPublishedAt() ?>
      <?php echo link_to($headline-
>getTitle(),'news/show?id='.$headline->getId()) ?>
    </li>
  <?php endforeach ?>
  </ul>
</div>
```

Ten komponent wyświetli 5 najnowszych wiadomości z bazy newsów.

Za każdym razem, kiedy chcemy wywołać komponent piszemy:

Listing 2.33 Wywołanie komponentu

```
<?php include_component('module', 'nazwakomponentu') ?>
```

Tak jak do partiali tak i do komponentu można dodatkowo przesłać pewne zmienne. Sprawę wyczerpuje przykład z listingu 2.34:

Listing 2.34 Zmienne w wywołaniu komponentu

```
<?php include_component('news', 'nazwakomponentu', array('foo' =>
'bar')) ?>
```
[13]

2.9. Warstwa modelu

Większość logiki biznesowej aplikacji internetowej opiera się na modelu danych. W symfony model danych domyślnie korzysta z warstwy mapowania obiektowo-relacyjnego (ORM) zapewnianą przez projekt Propel[14]. Dzięki temu narzędziu dostęp do danych zapisanych w bazie danych i ich modyfikacja odbywa się poprzez obiekty.

[13] Fragmnety kodu - „Inside The Viev Layer” – „T*he Definitive Guide to Symfony*”.
[14] http://propel.phpdb.org

Nigdy nie operuje się bezpośrednio na bazie danych, co pozwala na zachowanie wysokiego poziomu abstrakcji i przenośności. Podstawowym założeniem symfony jest to, że w ramach jednego projektu wszystkie aplikacje dzielą jeden model danych. Model jest częścią wspólną dla całego projektu, a podział na aplikacje powinien zależeć przede wszystkim od reguł biznesowych. Dlatego też pliki modelu są oddzielone od aplikacji i przechowywane są w osobnym katalogu *lib/model* katalogu głównego projektu. Aby stworzyć obiektowy model danych dla naszej aplikacji potrzebny jest nam schemat całej bazy danych. ORM potrzebuje opisu modelu relacyjnego zawartego w pliku xml lub yml. W schemacie definiuje się tabele, relacje, oraz kolumny i ich charakterystykę. Schemat ten znajduje się w katalogu config głównego korzenia projektu. Przykładowy schemat bazy z użyciem yml i xml

Listing 2.35 Schemat małej bazy danych składającej się z dwóch tabel w oparciu o standard yml[15]

```
propel:
  blog_article:
    _attributes: { phpName: Article }
    id:
    title:       varchar(255)
    content:     longvarchar
    created_at:
  blog_comment:
    _attributes: { phpName: Comment }
    id:
    article_id:
    author:      varchar(255)
    content:     longvarchar
    created_at:
```

[15], [16] Fragmenty schematów ORM- „Inside The Model Layer" – „T*he Definitive Guide to Symfony*".

Listing 2.36 Schmat oparty o standard xml [16]

```
<?xml version="1.0" encoding="UTF-8"?>
 <database name="propel" defaultIdMethod="native" noXsd="true"
package="lib.model">
    <table name="blog_article" phpName="Article">
      <column name="id" type="integer" required="true"
primaryKey="true"autoIncrement="true" />
      <column name="title" type="varchar" size="255" />
      <column name="content" type="longvarchar" />
      <column name="created_at" type="timestamp" />
    </table>
    <table name="blog_comment" phpName="Comment">
      <column name="id" type="integer" required="true"
primaryKey="true"autoIncrement="true" />
      <column name="article_id" type="integer" />
      <foreign-key foreignTable="blog_article">
        <reference local="article_id" foreign="id"/>
      </foreign-key>
      <column name="author" type="varchar" size="255" />
      <column name="content" type="longvarchar" />
      <column name="created_at" type="timestamp" />
    </table>
 </database>
```

Model danych nie jest uzależniony od rodzaju bazy danych jaką użyjemy. Minimalne informacje jakie są potrzebne dla symfony, by mogła pracować z bazą danych to nazwa bazy danych, kody dostępu oraz typ bazy danych. Wszystkie te informację umieszczamy w pliku databases.yml zlokalizowanym w katalogu config.

Schemat jak na listingach 2.35 lub 2.36 jest użyta do budowy klas przez warstwę ORM. Odbywa się to automatycznie przez wywołanie komendy propel-build-model. Uruchomienie tej komendy spowoduje odpalenie silnika analizującego schemat oraz tworzącego podstawowe klasy modelu w katalogu lib/model/om. Dla każdej tabeli tworzone są klasy BaseNazwatabeli.php oraz BaseNazwatabeliPeer.php dodatkowo zostaną utworzone 2 klasy w katalogu lib/model: Nazwatabeli.php i NazwatabeliPeer.php. Wydaje się to dosyć sporą ilością plików dla jednej tabeli, ale klasy typu BaseNazwatabeli.php oraz Nazwatabeli.php są to klasy, w których znajdują się obiekty reprezentujące rekordy w bazie danych. Dają one dostęp do kolumn w danym rekordzie oraz do powiązanych rekordów.

Listing 2.37 Przykład dostępu do rekordu

```
$article = new Article();
...
$title = $article->getTitle();
```

W klasach BaseNazwatabeliPeer.php oraz NazwatabeliPeer.php zawarte są statyczne metody do operowania na rekordach z bazy danych. Są to metody do pozyskania obiektów, nadpisywania oraz usuwania rekordów.

Listing 2.38 Pozyskiwanie obiektów modelu

```
$articles = ArticlePeer::retrieveByPks(array(123, 124, 125));
```

Pobieranie, zapisywanie i usuwanie danych

Każda z klas służących do obsługi konkretnej tabeli zawiera zespół konstruktorów, akcesorów oraz mutatorów związanych z dana tabelą oraz kolumnami. Metody new, getXXX() oraz setXXX() pomagają tworzyć obiekty oraz dają dostęp do ich właściwości. Poniżej przykład :

Listing 2.39 Działanie na obiektach modelu

```
$article = new Article();
$article->setTitle('My first article');
$article->setContent('This is my very first article.\n Hope you enjoy
it!');

$title   = $article->getTitle();
$content = $article->getContent();
```

Samo nadanie elementowi wartości przy użyciu metody set nie powoduje jego zapisu. Wszystkie operacje zapisu na obiekcie reprezentującym rekord muszą być zakończone użyciem funkcji save() na tym obiekcie. Usuwanie też należy do banalnie prostych

czynności. Gdy juz wybierzemy obiekt lub obiekty reprezentujące dany rekord, który chcemy usunąć wykonujmy na nim metodę delete().

2.10. Linki i system „Routingu”

Linki i adresy URL są unikalnymi punktami wejść aplikacji. Zastosowanie pomocników (helpers) w szablonie umożliwia całkowitą separację pomiędzy sposobem działania adresów URL, a jego widokiem. Nazywa się to routing. Routing jest mechanizmem, który zamienia adresy URL, aby miały bardziej użyteczną formę dzięki wykorzystaniu systemu przyjaznych lików oraz tłumaczy je tak, aby żądanie mogło trafić do odpowiedniej akcji i modułu.

Adresy URL przenoszą informacje z przeglądarki do serwera wymagane do podjęcia akcji utworzonej przez użytkownika. Na przykład, tradycyjny URL zawiera ścieżkę do skryptu i kilka parametrów niezbędnych do wykonania zapytania na przykład:

Listing 2.40 Przykładowy adres URL

```
http://www.example.com/web/controller/article.php?id=123456&format_code=6532
```

Taki rodzaj linku jest dosyć popularny w aplikacjach internetowych. Niestety jest to bardzo niebezpieczny sposób tworzenia URL-i. Na pierwszy rzut oka widać, że w linku podane są informacje na temat struktury projektu oraz bazy danych. Widać jaki skrypt jest wykonywany oraz jakie parametry przyjmuje do jego wykonania. W każdej chwili można zmienić parametry id co daje praktycznie nieograniczony interfejsu do przeglądania zawartości bazy danych. Można nawet spróbować zmienić nazwę skryptu i dostać się do innych funkcjonalności aplikacji. Takie linki przysparzają jeszcze kilku problemów. Jeżeli zmieni się nazwa skryptu lub jakiś inny element URL-a każdy link do tego adresu musi się zmienić. Wiąże się to w niektórych przypadkach z każdorazowymi bardzo kosztownymi zmianami w kontrolerze aplikacji.

Idea routingu jest związana z tym, że URL staje się częścią interfejsu aplikacji. Aplikacja może formatować URL-e i przekazywać informacje do użytkownika. Użytkownik może używać adresów do dostępu do zasobów aplikacji. Taka sytuacja jest możliwa w aplikacji symfony ponieważ URL prezentowany dla użytkownika końcowego nie jest związany z instrukcją serwera potrzebną do wykonania żądania, jest

jedynie związana z danym zasobem. Przykład adresu zrozumiałego przez system rootingu, który prowadzi do tego samego URL-a podanego w pierwszym przykładzie linka:

Listing 2.41 Przyjazny adres URL

```
http://www.example.com/articles/finance/2006/activity-breakdown.html
```

Jak widać nie ma tu żadnych elementów mówiących o tym jaki skrypt jest wykonywany. Dodatkowo takie formowanie linków bardzo sprzyja pracy robotów wyszukiwarek takich jak google. Dodatkowe profity z takiego sposobu prezentowania linków jakie nadmienia Fabien Potiencer w swej książce to:

- Takie URL przepisane do rożnego typu dokumentów są bardziej przejrzyste, czytelniejsze i łatwiejsze do zapamiętywania.
- Możliwość zmiany formatowania URL-a oraz parametrów action/parametr niezależnie, używając pojedynczej modyfikacji.
- Podczas reorganizacji wnętrza aplikacji, URL pozostaje ten sam dla danej aplikacji.
- Silniki wyszukiwania które normalnie opuszczają dynamiczne strony takie jak .php czy asp przeszukują je normalnie tak jak byłyby to strony statyczne.
- Ma to duże znaczenie dla bezpieczeństwa. Każdy nierozpoznany URL zostanie wysłany na stronę specyfikowaną przez użytkownika. Dodatkowo użytkownik nie może zobaczyć drzewa aplikacji przy jakichkolwiek próbach testowania URL-i.[17]

Cały sposób prezentacji URL-a i tłumaczenia go na odpowiednie akcje i parametry jest osiągany przez system routingu, który podlega bazowym wzorcom zaoferowanym przez symfony oraz jest modyfikowalny z poziomu konfiguracji aplikacji.

[17] *"The Definitive Guide to symfony"* - Links And The Routing System

Listing 2.42 Tłumaczenie adresów przez system rootingu

```
// Wewnętrzna składnia
<module>/<action>[?param1=value1][&param2=value2][&param3=value3]...

// Przykład wewnętrznej składni, która nigdy nie pojawia się
użytkownikowi
article/permalink?year=2006&subject=finance&title=activity-breakdown

// Zewnętrzny URL
http://www.example.com/articles/finance/2006/activity-breakdown.html[18]
```

Plik konfiguracyjny dla rootingu znajduje się w katalogu config danej aplikacji i nazywa się rooting.yml.

Komunikacja między stronami

Do komunikacji między stronami aplikacji symfony najlepiej używać helperów, o których wspominałem wcześniej. Oprócz tego, że są wygodne i zmniejszają ilość kodu dodatkowo sprzyjają jego przejrzystości i czystości. Oprócz znanego już link_to() przedstawię kilka innych przykładów wraz z różnymi opcjami wykorzystania:

[18] Przykłady linków oraz działania routingu *"The Definitive Guide to symfony"* - Links And The Routing System

Listing 2.43 Przykłady pomocników (helperów) używanych do nawigacji[19]

```
// Hyperlink on a string
<?php echo link_to('my article',
'article/read?title=Finance_in_France') ?>
 => <a href="/routed/url/to/Finance_in_France">my article</a>

// Hyperlink on an image
<?php echo link_to(image_tag('read.gif'),
'article/read?title=Finance_in_France') ?>
 => <a href="/routed/url/to/Finance_in_France"><img
src="/images/read.gif" /></a>

// Button tag
<?php echo button_to('my article',
'article/read?title=Finance_in_France') ?>
 => <input value="my article"
type="button"onclick="document.location.href='/routed/url/to/Finance_i
n_France';" />

// Form tag
<?php echo form_tag('article/read?title=Finance_in_France') ?>
<form method="post" action="/rout



<?php echo link_to('delete item', 'item/delete?id=123', 'confirm=Are
you sure?') ?>
 => <a onclick="return confirm('Are you sure?');"
       href="/routed/url/to/delete/123.html">delete item</a>

<?php echo link_to('add to cart', 'shoppingCart/add?id=100',
'popup=true') ?>
 => <a onclick="window.open(this.href);return false;"
       href="/fo_dev.php/shoppingCart/add/id/100.html">add to cart</a>

<?php echo link_to('add to cart', 'shoppingCart/add?id=100', array(
   'popup' => array('popupWindow',
'width=310,height=400,left=320,top=0')
)) ?>
 => <a
onclick="window.open(this.href,'popupWindow','width=310,height=400,lef
t=320,top=0');return false;"
       href="/fo_dev.php/shoppingCart/add/id/100.html">add to cart</a>
```

[19] Tabela najpopularniejszych helperów - *"The Definitive Guide to symfony"* - Links And The Routing System

2.11. Generatory

Większość aplikacji opiera się o pewien model danych umieszczonych w bazie. Framework symfony posiada narzędzie, które pozwala w sposób automatyczny stworzyć moduły odpowiedzialne za operacje na danych tabelach z bazy. W razie potrzeby framework umożliwia stworzenia w pełni funkcjonalnego panelu administracyjnego dla naszej aplikacji.

Generacja kodu na podstawie modelu

Operacje na danych w aplikacji webowej zwykle opierają się na działaniach takich jak dodawanie, wyświetlanie, edycja oraz usuwanie. Jest to tak rozpowszechniony sposób postępowania z danymi, iż otrzymał on swoją nazwę : CRUD czyli Create (tworzenie) Retrive (pobranie), Update (edycja, nadpisanie), Delete (usuwanie). Wiele funkcjonalności większości serwisów internetowych opiera się właśnie na takich działaniach. Przykładem może być umieszczanie newsów na portalach informacyjnych, tworzenie komentarzy do artykułów, czy też zamieszczanie ogłoszeń.

Symfony dzięki narzędziom propel-init-crud, propel-generate-crud, propel-init-admin pozwala na stworzenie kompletnego modułu odpowiedzialnego za wyżej wymienione operacje. Jedynym warunkiem jaki musi być spełniony jest poprawne zdefiniowanie modelu czyli schematu tabeli w bazi danych. Użycie generatorów wygląda jak na listingu 2.44

Listing 2.44 Komenda wywołująca generowanie modułu

```
> symfony <TASK_NAME> <APP_NAME> <MODULE_NAME> <CLASS_NAME>
```

Podczas rozwoju aplikacji generacja kodu może zostać wykorzystana do dwóch różnych celów:

Scaffolding – tworzy prostą strukturę dla danej tabeli, która pozwala w jak najprostszy i jak najbardziej oszczędny sposób manipulować zawartością tabeli. Jest to doskonały sposób na stworzenie szkieletu funkcjonalności dla danego zagadnienia związanego z jedną tabelą. Scaffolding stosuje się przeważnie w początkowych fazach projektowania. Stworzony dzięki temu narzędziu moduł można by nazwać prototypem.

Na takim module możemy dokonywać różnorakich zmian. Począwszy od wyglądu formularzy po sposoby wykonywania operacji na danych, czy też samej walidacji wprowadzanych danych. Saffolding można również wykorzystać do stworzenia szybkiego interfejsu służącego wypełnieniu bazy danych danymi potrzebnymi podczas tworzenia aplikacji lub testowania aplikacji.

By utworzyć scoffloading dla danej klasy modelu wystarczy wpisać komendę:

Listing 2.45 generowanie scoffloadnigu

```
symfony propel-generate-crud mojaaplikacja modulename baseclassname
```

Uruchamiając komendę z listingu 2.45 symfony analizuje schemat zawarty w pliku Schema.xml bądź Schema.yml oraz generuje odpowiednie szablony i akcje:

- List: jest to standardowy i startowy moduł dla danej tabeli. Wyświetla on wszystkie rekordy zawarte w tabeli
- Show: Jest to akcja wyświetlająca detale danego rekordu po kliknięciu na identyfikator w widoku list
- Edit : do tego widoku możemy się dostać z poziomu detali, czyli show. Tu możemy edytować zawartość rekordu lub go usunąć.

Poniższy listing prezentuje wszystkie elementy utworzone podczas scoffoldingu:

Listing 2.46 Spis elementów powstałych podczas scoffoldingu[20]

```
// In actions/actions.class.php
index           // Forwards to the list action below
list            // Displays the list of all the records of the table
show            // Displays the lists of all columns of a record
edit            // Displays a form to modify the columns of a record
update           // Action called by the edit action form
delete           // Deletes a record
create           // Creates a new record

// In templates/
editSuccess.php  // Record edition form (edit view)
listSuccess.php  // List of all records (list view)
showSuccess.php  // Detail of one record (show view)
```

[20] *"The Definitive Guide to symfony"* - Generators

Jak widać taki generator doskonale przyspiesza nam tworzenie podstawowej logiki do operacji na bazie danych. Dla niewymagających, tak wygenerowany kod, może być wystarczającym i wyczerpującym elementem funkcjonalności systemu. Może również posłużyć jako baza do rozwinięcia na niej dodatkowych zadań. Gdy w systemie mamy do czynienia z zarządzaniem dużą ilością danych w rożnych modułach staje się to nieocenione dla szybkości produkcji systemu jak i również integralności interfejsu.

Panele administracyjne – są to bardziej wyspecjalizowane i skomplikowane interfejsy do manipulacji danymi. Główną cechą, która odróżnia je od scaffoldingu jest to, że nie tworzą one modułu pozwalającego na ręczną modyfikację. Wszystko opiera się o plik konfiguracyjny generator.yml, w którym umieszcza się informacje na temat jakie dane oraz jakie funkcje mają być wykorzystane do tworzenia manipulacji rekordem. Cała logika działania panelu jest generowana automatycznie podczas uruchomienia w cachu aplikacji. Pozwala to na dowolną konfiguracje panelu w każdej chwili. Panele administracyjne oprócz tego, że posiadają już szczątkową grafikę, tworzą również automatycznie system sortowania oraz filtrowania danych. Aby utworzyć podstawowy panel administracyjny korzystamy z komendy podanej w listingu 2.47

Listing 2.47 budowanie modułu administracji dla danej tabeli

```
symfony propel-generate-admin mojaaplikacja modulename baseclassname
```

Listing 2.48 Plik konfiguracyjny dla przykładowego modułu

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Article
    theme:            default

    fields:
      author_id:      { name: Article author }

    list:
      title:          List of all articles
      display:        [title, author_id, category_id]
      fields:
        published_on: { params: date_format='dd/MM/yy' }
      layout:         stacked
      params:         |
        %%is_published%%<strong>%%=title%%</strong><br /><em>by %%author%%
        in %%category%% (%%published_on%%)</em><p>%%content_summary%%</p>
      filters:        [title, category_id, author_id, is_published]
      max_per_page:   2

    edit:
      title:          Editing article "%%title%%"
      display:
        "Post":       [title, category_id, content]
        "Workflow":   [author_id, is_published, created_on]
      fields:
        category_id:  { params: disabled=true }
        is_published: { type: plain}
        created_on:   { type: plain, params: date_format='dd/MM/yy' }
        author_id:    { params: size=5 include_custom=>> Choose an author << }
        published_on: { credentials:  }
        content:      { params: rich=true tinymce_options=height:150 }
```

2.12. Testowanie

Tworzenie różnorakich aplikacji komputerowych, desktopowych, czy też webowych nieodłącznie wiąże się z ich testowaniem. W aplikacji testuje się praktycznie każdy aspekt jej działania. Czasem testy i naprawa błędów, które są wychwycone pochłania większą część czasu tworzenia aplikacji. Symfony udostępnia zestaw narzędzi, które wspomagają testy zautomatyzowane, które nie tylko oszczędzają nam wiele czasu, ale i znacznie wpływają na jakość tworzonej aplikacji.

Zautomatyzowane testy

Programowanie aplikacji webowych wymusza częste zmiany dotyczące jej wymagań. Zdarzą się, iż w miarę rozwoju lub już eksploatacji pojawiają się nowe problemy do rozwiązania. Związane są one zwykle z optymalizacją, dodaniem nowych funkcjonalności sugerowanych przez użytkowników i tym podobne. Częste zmiany w kodzie lub jego faktoryzacja może doprowadzić do pojawienia się błędów. Każdorazowe tworzenie nowych testów to mordercza praca. W takich przypadkach zbawienne wydaje się użycie testów zautomatyzowanych. Testy zautomatyzowane zapobiegają przypadkowym błędom w kodzie oraz zmuszają programistów do tworzenia ich w jednym ustalonym formacie. Dobrze stworzony zestaw testów może czasem służyć jako dokumentacja samej aplikacji. Opis wartości wejściowych oraz wyjściowych w testach doskonale obrazuje funkcjonalność danej metody co praktycznie wyczerpuje jej udokumentowanie. W symfony wykorzystywane są dwa rodzaje testów zautomatyzowanych: testy jednostkowe i funkcjonalne.

Testy jednostkowe

Testy jednostkowe są to testy sprawdzające poprawność wejścia i wyjścia określonych fragmentów kodu. Taka sytuacja dotyczy przeważnie funkcji i metod. Dla jednej metody czy funkcji tworzymy wiele testów sprawdzających szczególne przypadki ich działania.

Istnieje wiele framework-ów służących do tworzenia testów jednostkowych w PHP, jak chociażby najbardziej znane PhpUnit i SimpleTest. Symfony posiada własny, zwany Lime. Opiera się on na bibliotece języka Perl Test::More i jest zgodny z TAP, co oznacza że wyniki testów są wyświetlane tak, jak określono to w protokole TAP (Test Anything Protocol), zaprojektowanym w celu uzyskania lepszej czytelności wyników testów. [21]

Lime składa się z jednego pliku lime.php i jest w pełni autonomicznym nie posiadającym żadnych zależności systemem testowania. Testy jednostkowe w symfony wykorzystujące Lime umieszczone są w katalogu test/unit każdej aplikacji. Testy te są

[21] *"The Definitive Guide to symfony"* – Unit and functional Testing - tłumaczenie

prostymi plikami PHP, których nazwy kończą się na Test.php. Poniżej znajduję się przykład kilku testów jednostkowych dla funkcji strlower() przytoczony przez autora książki *The Definitive Guide to Symfony. Listing 2.49*

Test zaczyna się od inicjalizacji obiektu lime_test. Każdy z testów jest uruchomianiem pewnej metody obiektu lime_test, która za argument przyjmuje wywołanie testowanej metody z rocznymi parametrami oraz łańcuch znaków służący za komentarz.

Listing 2.49 Zbiór testow lime

```
[php]
<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');
require_once(dirname(__FILE__).'/../../lib/strtolower.php');

$t = new lime_test(7, new lime_output_color());

// strtolower()
$t->diag('strtolower()');
$t->isa_ok(strtolower('Foo'), 'string',
    'strtolower() zwraca typ string');
$t->is(strtolower('FOO'), 'foo',
    'strtolower() zamienia duże litery na małe');
$t->is(strtolower('foo'), 'foo',
    'strtolower() zostawia małe litery niezmienione');
$t->is(strtolower('12#?@~'), '12#?@~',
    'strtolower() zostawia znaki spoza alfabetu niezmienione');
$t->is(strtolower('FOO BAR'), 'foo bar',
    'strtolower() zostawia odstępy niezmienione'); ????
$t->is(strtolower('FoO bAr'), 'foo bar',
    'strtolower() radzi sobie z napisem zawierającym litery o różnej
wielkości');
$t->is(strtolower(''), 'foo',
    'strtolower() zamienia puste łańcuchy w napis foo');
```

Uruchomiony test wyświetla wyniki w konsoli bardzo wyraźnie sygnalizując czy pojedynczy test zakończył się pomyślnie. Listing 2.50

Listing 2.50 Wynik testu

```
> symfony test-unit strtolower

1..7
# strtolower()
ok 1 - strtolower() zwraca typ string
ok 2 - strtolower() zamienia duże litery na małe
ok 3 - strtolower() zostawia małe litery niezmienione
ok 4 - strtolower() zostawia znaki spoza alfabetu niezmienione
ok 5 - strtolower() zostawia odstępy niezmienione
ok 6 - strtolower() radzi sobie z napisem zawierającym litery o różnej
wielkości
not ok 7 - strtolower() zamienia puste łańcuchy w napis foo
#     Failed test (.\batch\test.php at line 21)
#            got: ''
#       expected: 'foo'
# Looks like you failed 1 tests of 7.
```

Testy funkcjonalne

Testy funkcjonalne służą do weryfikacji działania części aplikacji. Symulują sesję użytkownika, wysyłają żądania i sprawdzają elementy obecne w odpowiedzi W testach funkcjonalnych badany jest scenariusz odpowiadający danemu przypadkowi użycia. Przykładem może być działanie całej akcji z jej generowaniem i pobieraniem z cache'a.

W przypadku bardziej skomplikowanych interakcji, powyższe sposoby mogą okazać się niewystarczające. Operacje z użyciem AJAX, dla przykładu, wymagają przeglądarki internetowej, aby wykonać JavaScript, więc automatyczne testowanie wymaga dodatkowego narzędzia. Ponadto efekty wizualne mogą być walidowane tylko przez człowieka.

Symfony dostarcza specjalnego obiektu, zwanego sfBrowser, który symuluje działanie przeglądarki podłączonej do aplikacji symfony bez konieczności korzystania z serwera - i wolnej od opóźnień związanych z transmisją przez HTTP. Obiekt ten daje dostęp do istotnych części każdego żądania (request, sesji, context, response). Symfony dostarcza również klasę będąca jej rozszerzeniem, sfTestBrowser, zaprojektowaną specjalnie do testów funkcjonalnych i posiadającą, oprócz wszystkich możliwości klasy sfBrowser, kilka eleganckich metod służących do sprawdzania asercji. [22] Test funkcjonalny

[22] *"The Definitive Guide to symfony"* – Unit and functional Testing - tłumaczenie

inicjalizuje obiekt typu browser, wysyła żądanie i sprawdza czy wszystkie wymagane elementy obecne są w odpowiedzi. Symfony przy generowaniu każdego modułu automatycznie tworzy prosty test funkcjonalny. Test ten wysyła żądanie do domyślnej akcji modułu sprawdza status odpowiedzi, poprawność routingu oraz obecność pewnych fraz w samej treści odpowiedzi. Dla modułu o nazwie modul utworzy on plik o nazwie modulActionTest.php, który wygląda tak jak na listingu 2.51

Listing 2.51 Test funkcjonalny dla modułu

```
php]
<?php

include(dirname(__FILE__).'/../../bootstrap/functional.php');

// Create a new test browser
$browser = new sfTestBrowser();
$browser->initialize();

$browser->
  get('/modul/index')->
  isStatusCode(200)->
  isRequestParameter('module', 'modul')->
  isRequestParameter('action', 'index')->
  checkResponseElement('body', '!/To jest strona tymczasowa/')
;
```

Aby uruchomić taki test funkcjonalny używamy polecenia test-functional wpisując je linii poleceń Symfony listing 2.52

Listing 2.52 Wynik testu funkcjonalnego

```
> symfony test-functional frontend foobarActions

# get /comment/index
ok 1 - status code is 200
ok 2 - request parameter module is foobar
ok 3 - request parameter action is index
not ok 4 - response selector body does not match regex /This is a
temporary page/
# Looks like you failed 1 tests of 4.
1..4
```

2.13. Uruchomienie i konfiguracja projektu symfony

Projekt aplikacji webowej wykorzystujący symfony jest oparty o pewną strukturę katalogów. Jest to drzewiasta struktura posiadająca rożne prawa dostępu do odpowiednich katalogów. Silnik symfony całkowicie automatyzuje tworzenie projektów. Aby stworzyć projekt należy stworzyć katalog i wykonać jedną komendę w linii poleceń listing 2.53

Listing 2.53 Tworzenie projektu symfony

```
> mkdir ~/myproject
> cd ~/myproject
> symfony init-project myproject
```

Samo polecenie ./symfony musi być zawsze uruchomione w katalogu z projektem, gdyż wszystkie zadania wykonywane przez te komendę są powiązane z projektem przez katalog w którym się znajduje.

Symfony utworzy strukturę katalogów opisaną w rozdziale 2.6:

Listing 2.54 Struktura projektu symfony

```
apps/
batch/
cache/
config/
data/
doc/
lib/
log/
plugins/
test/
web/
```

Projekt nie jest jeszcze gotowy do przeglądania, ponieważ wymaga co najmniej jednej aplikacji. Polecenie z listingu 2.55.

Listing 2.55 Tworzenie nowej aplikacji w projekcie

```
> symfony init-app myapp
```

Tworzy nam pierwszą aplikację o nazwie myapp i strukturze opisanej w rozdziale 2.6

Pozostaje nam jeszcze skonfigurowanie serwera wirtualnego, aby aplikacja była widoczna w naszej przeglądarce. Możemy to zrobić w dwojaki sposób.

- Konfigurując odpowiednio serwer stron WWW (w naszym przypadku apache)

  Przykładowa konfiguracja w pliku httpd.conf

Listing 2.56 przykładowa konfiguracja wirtualnego hosta dla aplikacji Symfony[23]

```
<VirtualHost *:80>
  ServerName myapp.example.com
  DocumentRoot "/home/user/myproject/web"
  DirectoryIndex index.php
  Alias /sf /$sf_symfony_data_dir/web/sf
  <Directory "/$sf_symfony_data_dir/web/sf">
    AllowOverride All
    Allow from All
  </Directory>
  <Directory "/home/user/myproject/web">
    AllowOverride All
    Allow from All
  </Directory>
</VirtualHost>
```

- Umieszczenie linka symbolicznego do katalogu /Web naszej aplikacji, w katalogu domyślnym, dla serwera stron WWW. W naszym przypadku jest to katalog /var/www.

Listing 2.57 inny sposób na udostępnienie aplikacji

```
ln -s /home/user/myproject/web /var/www/linkname
```

[23] *"The Definitive Guide to symfony"* – Running Symfony

2.14. Podsumowanie

Podsumowując informacje na temat frameworka symfony przyjrzyjmy się jak wyglądałby przykład prostego programu, który przytoczyłem przy okazji omawiania wzorca mvc w środowisku symfony.

Listing 2.62 Plik kontrolera

```
<?php
class weblogActions extends sfActions
{
  public function executeList()
  {
    $this->posts = PostPeer::doSelect(new Criteria());
  }
}

?>
```

Powyżej plik kontrolera znajdujący się według schematu budowy aplikacji symfony w katalogu `myproject/apps/myapp/modules/weblog/actions/`

Do tego: plik widoku listSucces.php widoczny na listingu 2.63.

Listing 2.63 Plik widoku

```
<h1>List of Posts</h1>
<table>
<tr><th>Date</th><th>Title</th></tr>
<?php foreach ($posts as $post): ?>
  <tr>
    <td><?php echo $post->getDate() ?></td>
    <td><?php echo $post->getTitle() ?></td>
  </tr>
<?php endforeach; ?>
</table>
```

W katalogu:

```
myproject/apps/myapp/modules/weblog/templates/listSuccess.php
```

Przedstawię jeszcze layout.php jako część widoku wspólną dla wszystkich akcji tej aplikacji.

Listing 2.64 Plik widoku wspólny dla całej aplikacji

```
<html>
  <head>
    <?php echo include_title() ?>
  </head>
  <body>
    <?php echo $sf_data->getRaw('sf_content') ?>
  </body>
</html>
```

Cała warstwa modelu jest wygenerowana automatycznie na podstawie danych ze schemy.

Jak widać dzięki wykorzystaniu symfony została zachowana przejrzystość kodu. Znacznie zmniejszyła się ilość potrzebnego kodu do zaprogramowania przez programistę, a co za tym idzie- zmniejszył się czas jego produkcji oraz prawdopodobieństwo wystąpienia błędu.

3. *Wymagania projektowanego systemu*

Dobra polityka produkowania systemów komputerowych opiera się najczęściej o pewien model. Najpopularniejszym z takich modeli jest model kaskadowy. Zakłada on kilku etapowy podział pracy nad systemem:

- Planowanie systemu (w tym Specyfikacja wymagań)
- Analiza systemu (w tym Analiza wymagań i studium wykonalności)
- Projekt systemu (poszczególnych struktur itp.)
- Implementacja (wytworzenie kodu)
- Testowanie (poszczególnych elementów systemu oraz elementów połączonych w całość)
- Wdrożenie i pielęgnacja powstałego systemu.

Pierwszym i bardzo ważnym etapem produkcji system jest określenie wymagań. Determinują one każdy dalszy etap i jako punkt wyjścia nadają kierunek wszystkim pracą. Poniżej przedstawię spis wymagań dotyczących naszego systemu

3.1. Wymagania funkcjonalne

3.1.1. Podstawowe wymagania systemu

Podstawowym założeniem systemu jest gromadzenie i usystematyzowanie danych pomiarowych z roczników pomiarowych. Dane te podzielone są na 2 główne grupy:

- dane z pomiarów manualnych na posterunkach
- dane automatyczne

3.1.2. Warstwa dodawania danych

System zakłada dwojaki sposób zasilania bazy danych w pomiary.

- Schemat standardowy polegający na wpisaniu danych do generowanych formularzu.
- Automatyczny zasilający bazę danych przez export z plików XML.

Dla danych manualnych system ma generować formularze dla odpowiednich stacji pomiarowych uwzględniając zakres czasowy pomiaru oraz instrumentów pomiarowych obecnych na stacji.

Dane dla pomiarów automatycznych mają być pobierane przez parsowanie plików XML. Dla danych musi być przygotowany słownik z opisem instrumentów pomiarowych, ponieważ pliki eksportowe nie zawierają tych danych. Słownik można będzie uzupełnić za pomocą odpowiedniego interfejsu lub pobrać ze źródła zewnętrznego.

3.1.3. **Warstwa edycji danych**

Edycja danych ma się odbywać w jak najmniej uciążliwy sposób. Moduł edycji powinien być zintegrowany z modułem dodawania oraz modułem wyszukiwania, by w szybki sposób można było nadpisać lub dopisać do istniejących danych. Do edycji należy wykorzystać jeden z dostępnych mechanizmów AJAX- a mianowicie In-place Edit.

3.1.4. **Wyszukiwanie oraz prezentacja danych**

- Moduł wyszukiwania

Moduł musi w intuicyjny i prosty sposób pozwolić na wyłowienie poszukiwanych danych oraz posiadać dość elastyczny sposób określania kryteriów. Kryteria jakie muszą być uwzględnione to:

  - Czas pomiaru
  - Stacja pomiarowa
  - Rodzaj pomiaru
- Prezentacja danych

Dane maja być prezentowane w dwojaki sposób. Pierwszym sposobem prezentacji jest wyświetlenie danych w oknie przeglądarki w postaci sformatowanych tabel oraz wykresów . Jako drugi sposób system ma pozwalać zaprezentować dane w formatach użytkowych takich jak XML , PDF oraz CSV.

3.1.5. **Bezpieczeństwo**

System nie ma posiadać wyszukanego systemu bezpieczeństwa. Wystarczającym zabezpieczeniem będzie zablokowanie dostępu postronnym użytkownika przez wprowadzenie systemu autentykacji. Dostęp do wszystkich modułów będzie możliwy jedynie po wprowadzeniu odpowiednich danych w formularzu logowania, który będzie startową stroną po otwarciu systemu.

3.1.6. **Wygląd oraz interfejs**

System ma być przejrzysty oraz intuicyjny. Barwy użyte w interfejsie maja być stonowane, pastelowe. Ważniejsze informacji , komunikaty o błędach oraz podpowiedzi mają być wyświetlane w widoczny sposób. Mogą one kontrastować z ogólną kolorystyką.

Wygląd menu nawigacyjnego standardowy. Główne menu ma się znajdować w widocznym miejscu, najlepiej nagłówku strony. Menu poboczne po lewej stronie okna.

3.2. Wymagania niefunkcjonalne

3.2.1. **Użytkownicy**

Użytkownikami systemu będą osoby z podstawowymi umiejętnościami obsługi komputera

3.2.2. **Obsługa błędów**

Błędy, które mogą pojawić się w systemie będą sklasyfikowane w trzech typach:

- Błędy fatalne środowiska (krytyczne)
    - Błędy systemowe – brak pamięci
    - Błędy czytania źródła
    - Błędy wyświetlenia stron

  - Błędy związane z zasobami systemowymi i dyskowymi

Błędy te będą obsługiwane przez przerwanie działania skryptu oraz przez skierowanie na stronę błędu

- Błędy operacyjne (nie krytyczne)
  - Problem z zapisem do bazy
  - Problem z sesją użytkownika

Błędy będą obsługiwane przez wyświetlenie odpowiednich komunikatów użytkownikowi oraz umieszczenie informacji w logach.

- Błędy danych wejściowych
  - Wprowadzanie danych w niezgodnych formatach
  - Niezgodne z żądanym formaty plików
  - Próba otwarcia nieistniejących zasobów

Błędy tego typu będą walidowane na poziomie prowadzania lub wykonywania skryptów informując użytkownika o błędzie, sugerując prawidłowe wprowadzenie.

3.2.3. Wymagania sprzętowo systemowe

Podstawowym wymogiem dla systemu jest użycie narzędzi typu open source. Podstawowe z tych wymagań to:

- Użycie darmowej platformy jako oprogramowania serwerowego
  - Linux jako system operacyjny
  - Platforma programistyczna: framework symfony oparty o php5
  - Serwer stron WWW Apacze 2
  - Włączona obsługa mod_rewrite
  - Baza danych MySql
- Wymagania sprzętowe ograniczą się do posiadania odpowiedniej do ilości danych przestrzeni dyskowej oraz ilość pamięci pozwalająca na wykonywanie dość wymagających skryptów parsowania danych.

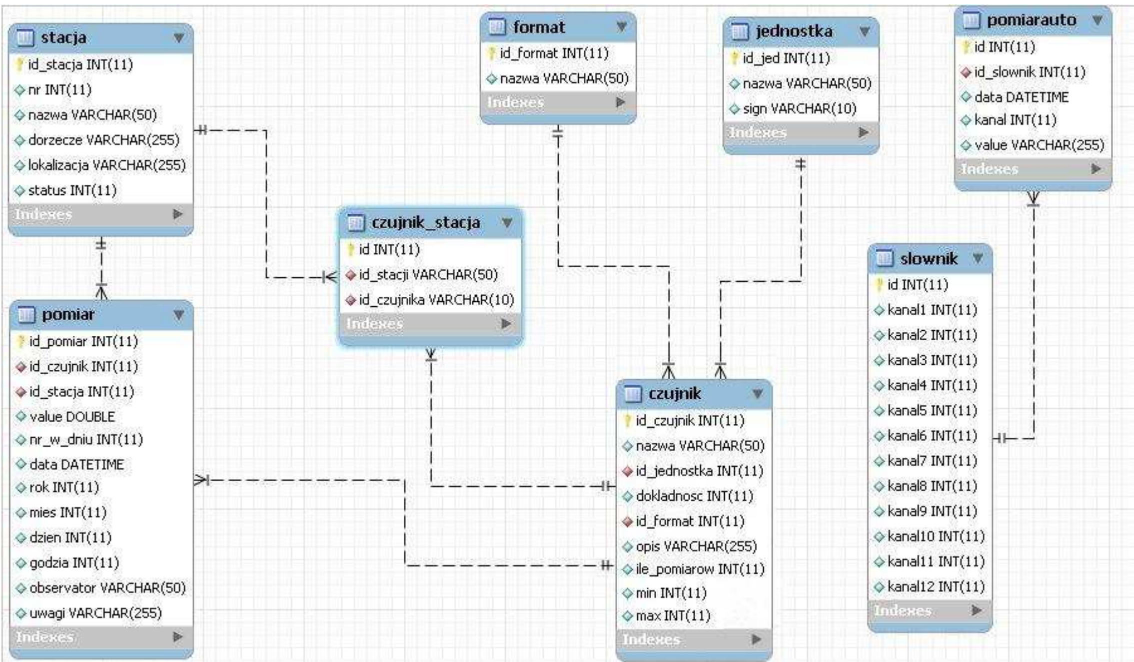
- Konfiguracja będzie następstwem wykorzystanego framework-a. Konfiguracja serwera stron WWW musi przewidzieć przedłużony czas działania skryptów co najmniej 60 sekund. Pamięć przeznaczona na skrypty to minimum 64 Mb.

4. Implementacja

Zaprojektowany system ma na celu gromadzenie, prezentacje danych hydrologicznych oraz prezentacje ich w odpowiednich formatach. System stworzony w oparciu o framework symfony jest rozproszonym systemem webowym pozwalającym na prace w sieci lokalnej ,intranetowej lub też z poziomu sieci web.

System TajFUNv.0.2 ma budowę modularną. Jest podzielony na kilka modułów z których każdy spełnia odrębną funkcjonalność. (rozdział 2.2). Poniżej przedstawiam strukturę bazy danych systemu.

4.1. Struktura bazy danych



Rys 4.1 Struktura bazy danych. [źródło: opracowanie własne]

Powyższy rysunek przedstawia strukturę bazy danych na której opiera się stworzony system. Baza danych jest bazą relacyjną. Najważniejsze tabele to pomiar oraz pomiarauto. Tabela pomiar służy do przechowywania pomiarów standardowych z dzienników natomiast tabela pomiarauto jest przystosowana dla pomiarów automatycznych o znacznie większej ilości rekordów.

4.2. Opis modułów:

System składa się z kilku modułów z których każdy prezentuje inną funkcjonalność. Poniżej przedstawiam listę zaimplementowanych modułów wraz z krótkim opisem działania. Następnie najważniejsze z nich opisze dokładniej wraz z podaniem przykładów działania.

- main – jest to moduł startowy, w którym ładowna jest strona domowa systemu z krótką informacją na temat systemu.

- security – jest to moduł odpowiedzialny za autentykację i dostęp do pozostałych modułów.

- czujniki – moduł odpowiedzialny za zarządzanie bazą czujników pomiarowych. Odpowiada za wypełnianie bazy danych czujnikami mierzącymi odpowiednie parametry wraz z tworzeniem jednostek pomiarowych oraz formatów czujników.

- stacje – moduł ten odpowiada za tworzenie stacji pomiarowej. Określanie jej odpowiednich parametrów oraz przyporządkowywanie jej odpowiednich czujników

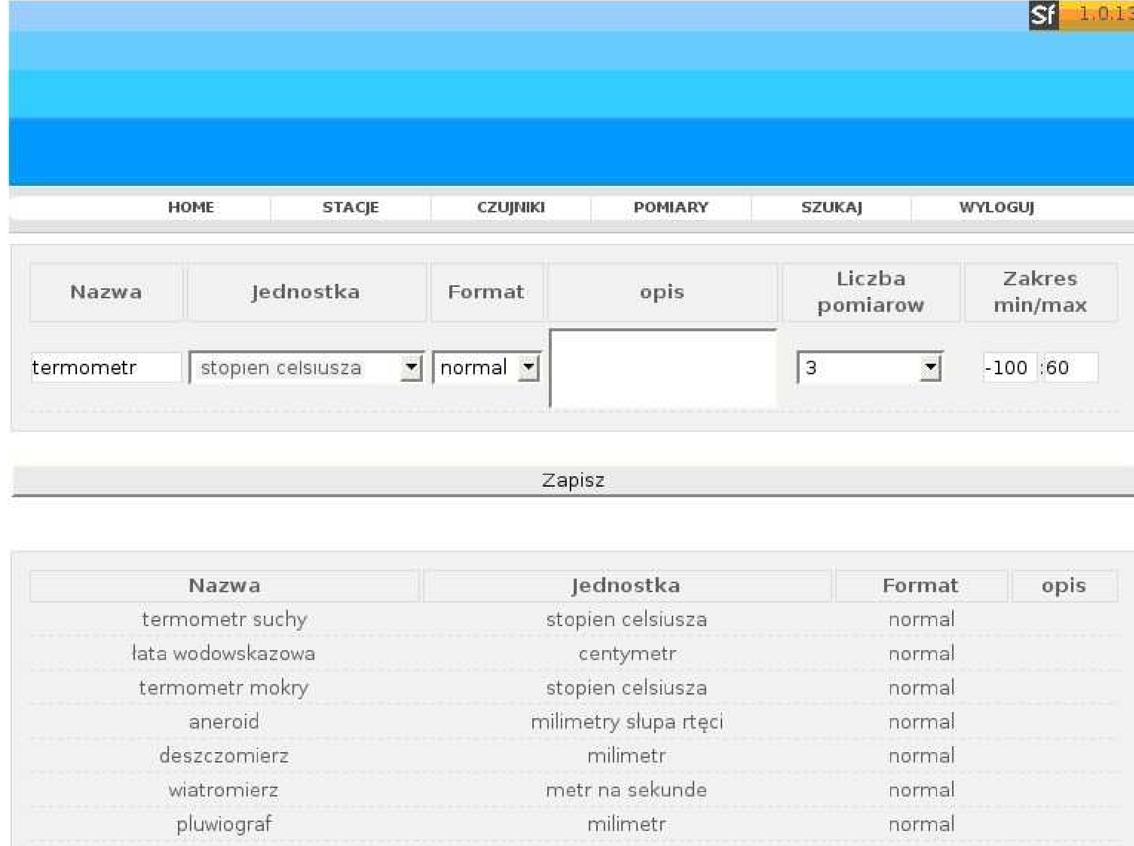
- pomiary – jest to moduł odpowiedzialny za uzupełnianie bazy danych pomiarami hydro-meteorologicznymi. Współpracuje on z modułem import, który umieszcza w bazie danych pomiary automatyczne.

- search – moduł wyszukania, który odpowiada z budowę interfejsu wyszukiwania oraz wyświetlanie rezultatów wyszukiwania. Posiada on również funkcjonalność prezentacji oraz exportu danych wyszukanych do użytecznych formatów.

- errors – moduł w którym znajdują się strony wyświetlające błędy krytyczne systemu.

4.3. Moduł czujniki.

Moduł ten służy do tworzenia oraz opisu urządzeń pomiarowych w systemie. Zadania jakie do niego należą to: utworzenie słownika jednostek pomiarowych oraz zbudowanie i nadanie właściwości urządzeniom pomiarowym. Cały proces rozpoczyna zdefiniowanie jednostek pomiarowych dla czujników. Po wypełnieniu formularza z nazwą oraz oznaczeniem jednostki klikamy w „Zapisz" co powoduje dodanie danych do bazy danych. Po dodaniu jednostek możemy przejść do utworzenia czujnika. Sytuacja jest podobna jak powyżej z tym, że mamy bardziej rozbudowany formularz do wypełnienia, w którym dodatkowo znajduje się pole wyboru wcześniej dodanych jednostek.

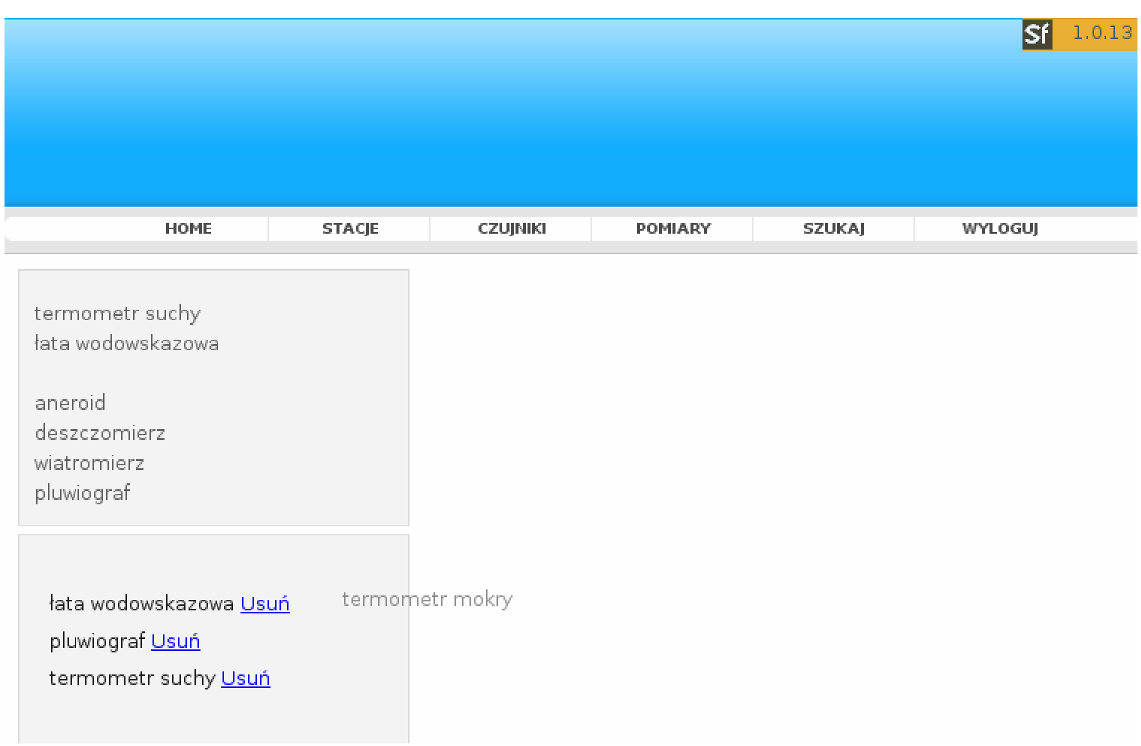


Rys 4.2 Tworzenie urządzenia pomiarowego. [źródło: opracowanie własne]

Rys 4.3 Tworzenie jednostek dla urządzeń pomiarowych. [źródło: opracowanie własne]

4.4. Moduł stacje.

Moduł odpowiada za tworzenie zbioru stacji pomiarowych z których odczytane dane będą przechowywane w systemie. Tworzenie odbywa się przez opisanie stacji informacjami takimi jak: nazwa, położenie, dorzecze oraz określenie czy na takiej stacji znajdują się pomiary automatyczne. Kolejnym etapem jest dodanie urządzeń pomiarowych do danej stacji. Odbywa się to dzięki prostej konstrukcji drag and drop (rys. 4.4). Z listy urządzeń pomiarowych typu manualnego zdefiniowanych w module czujniki wybieramy te, które znajdują się na stacji i przeciągamy je do okna danej stacji.

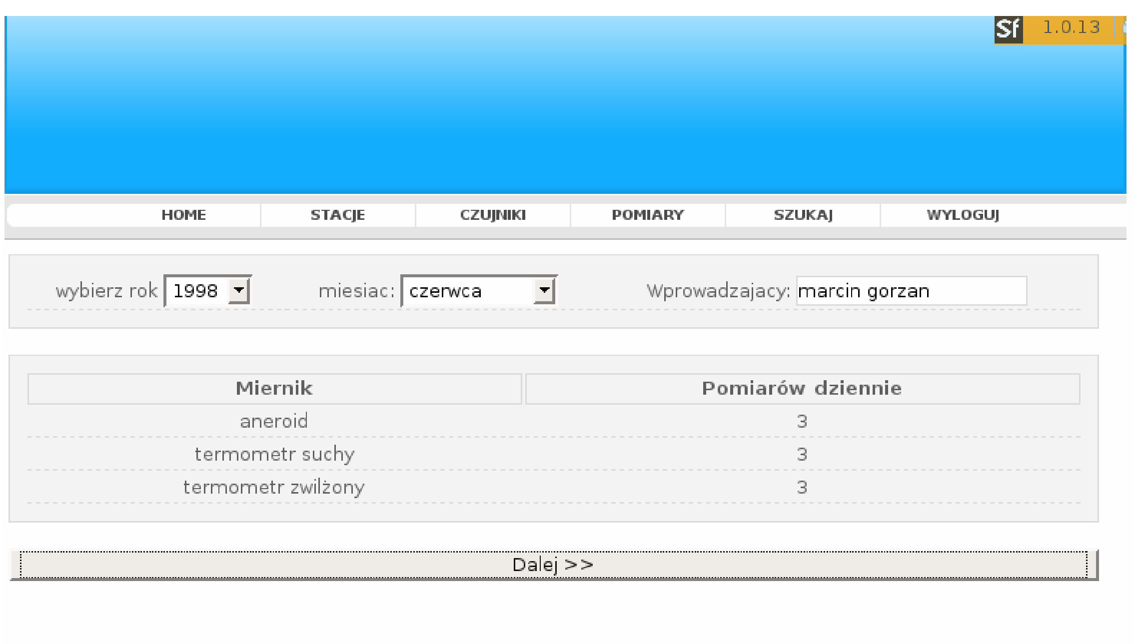


Rys 4.4 Uzupełnianie stacji urządzeniami pomiarowymi. [źródło: opracowanie własne]

Rys 4.5 Dodawanie stacji pomiarowych. [źródło: opracowanie własne]

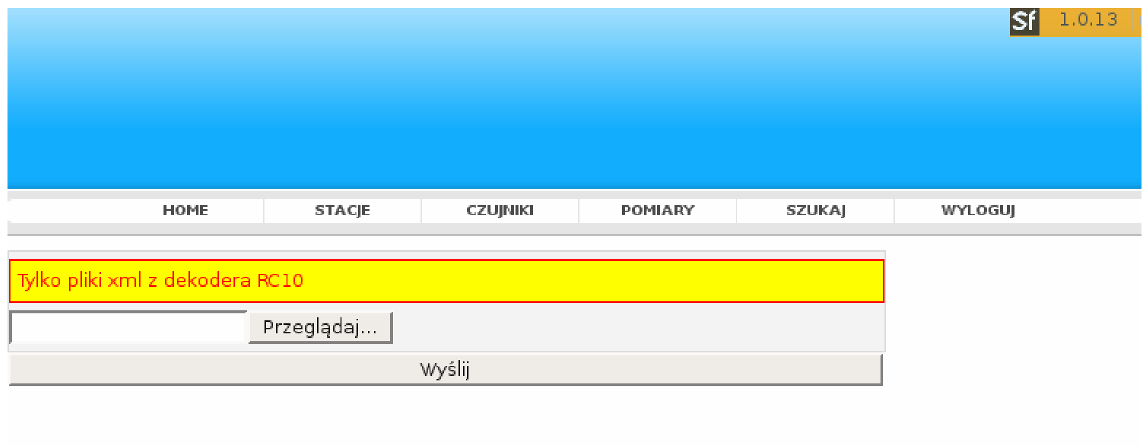
4.5. Moduł pomiary.

Ten moduł odpowiedzialny jest za uzupełnianie bazy pomiarami. Po wejściu do modułu mamy wybór: czy uzupełniamy dane z roczników, czy importujemy dane automatyczne. Przy pierwszym z wyborów generowana jest lista stacji. Po wyborze odpowiedniej stacji oraz określeniu ram czasowych pomiaru tworzony jest automatyczny formularz do uzupełniania pomiarów. Wygląd i budowa formularza zależy od ilości urządzeń pomiarowych jak i od ilości pomiarów w dniu. Przed wygenerowaniem formularza system sprawdza czy już dane o takich kryteriach nie zostały wprowadzone wcześniej. W razie wystąpienia takiej sytuacji pyta czy nadpisać istniejące dane czy może edytować / dopisać do istniejących. Cała akcja kończy się zapisem do bazy danych przez wciśniecie „Zapisz”. Oprócz danych jednostkowych system udostępnia możliwość przechowywania danych z pomiarów automatycznych zdekodowanych przez dekoder RC10 (Marcin Kowal, 2008). Cały proces polega na wysłaniu plików xml do systemu oraz wyparsowaniu odpowiednich danych wzbogacając ich o rodzaj mierzonego parametru oraz o lokalizacje. Wszystkie akcje dotyczące parsowania plików XML choć są związane z dodawaniem pomiarów umieszczone są w osobnym module ”import”.



Rys 4.7 Dodawanie pomiaru. [źródło: opracowanie własne]

Rys 4.8 Uzupełnianie formularza z pomiarami. [źródło: opracowanie własne]



Rys 4.9 Import pomiarów automatycznych. [źródło: opracowanie własne]

4.6. Moduł import.

Moduł ten odpowiada za parsowanie pliku z danymi za pomocą algorytmów zaimplementowanych oraz opisanych w pracy Justyny Kowal.

4.7. Moduł wyszukiwania.

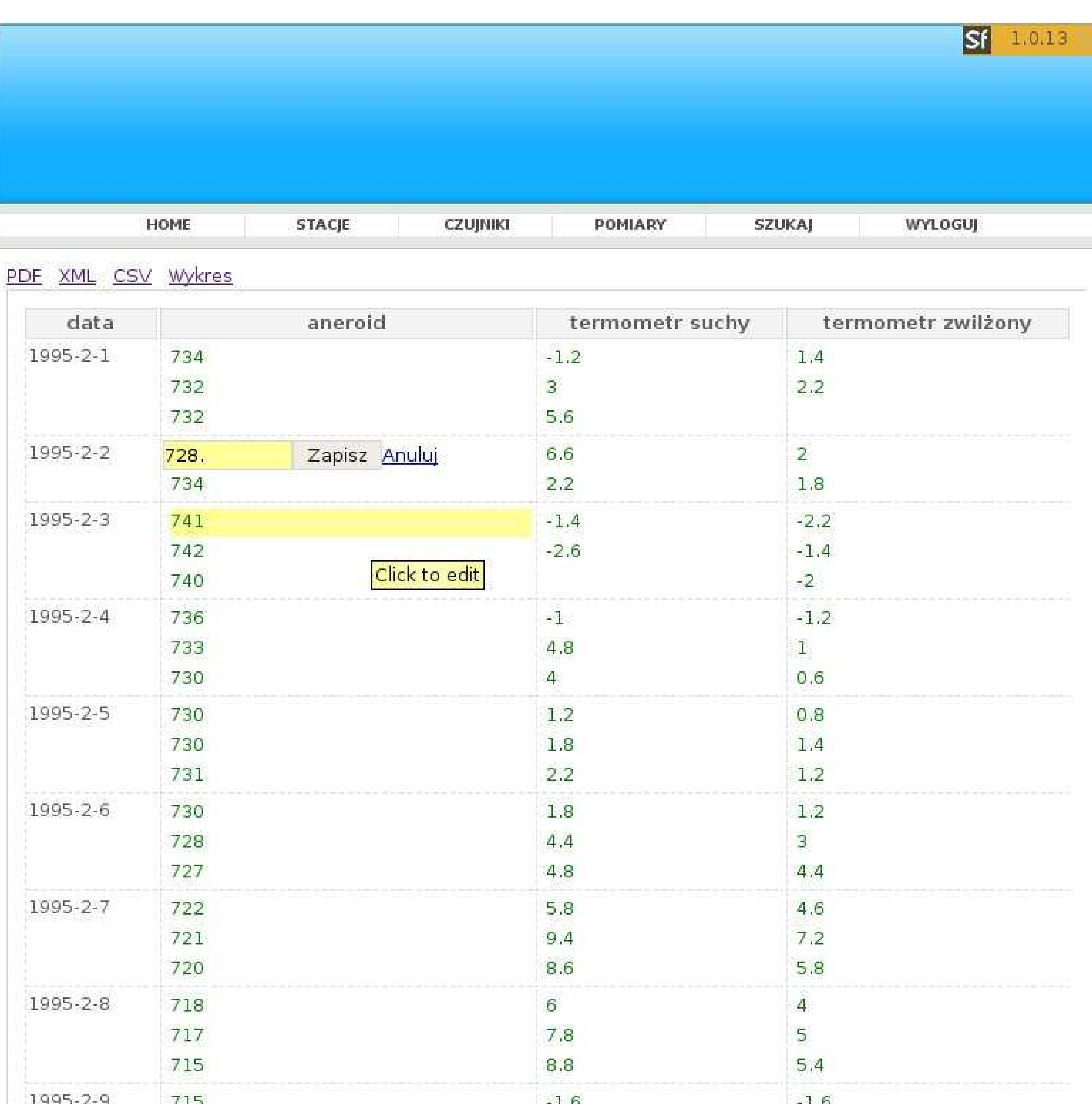
Moduł wyszukiwania jest bardzo prostą intuicyjną w obsłudze wyszukiwarką pomiarów. Po wejściu na moduł wyszukiwania mamy do dyspozycji pewne okna „kryterialne”, w których odpowiednio znajdują się lata, miesiące, stacje, oraz czujniki pomiarowe. W przypadku, gdy chcemy poszukać np. pomiaru temperatury powietrza na stacji w Porąbce, we wrześniu 1995 roku- przeciągamy konkretne elementy z okien wyboru do kontenera, po środku tworząc zespół kryteriów a końcowo zapytanie do bazy. Wszytko odbywa się dzięki prostemu i intuicyjnemu systemowi podnieś i upuść (drag and drop). Po wybraniu odpowiednich kryteriów pojawia nam się przycisk „Szukaj”. Minimalne kryteria jaki są przewidziane przez system to wybranie roku, stacji oraz miesiąca lub czujnika. W przeciwnym wypadku system mógłby mieć problemy z wyświetleniem dużych ilości danych. Rezultatem wyszukiwania jest tabelka bądź tabelki z danymi odpowiadającymi kryteriom. Z poziomu okna rezultatu możliwy jest eksport wyszukanych danych do kilku formatów oraz przedstawienie ich w postaci trójwymiarowych wykresów. Obsługa exportu znajduje się w osobnym module export. Który spełnia jeszcze jedną funkcje o czym poniżej. Dostępne formaty to XML, PDF, oraz CSV.

Moduł wyszukiwania zintegrowany jest z modułem edycji danych, co daje dosyć szybkie narzędzie do poprawy błędów lub aktualizacji danych. Wszystkie elementy tabeli rezultatu wyszukania tj. wartości pomiarowe w tej tabelce, są odrębnymi elementami pozwalającymi na edycje danej wartości. Przy wykorzystaniu technologii AJAX system pozwala zmienić wartość wyszukanych danych jedynie przez kliknięcie na daną wartość, wprowadzenie nowej i zapisanie. Edycja w miejscu (In place Edit) jest bardzo wygodnym i szybkim narzędziem do zmian i poprawek dokonywanych w pomiarach.

Mechanizm działania systemu edycji w miejscu przedstawiam poniżej.

Rys 4.10 Okno tworzenia kryteriów wyszukiwania. [źródło: opracowanie własne]

Moduł exportu oprócz przedstawienia danych w różnych formatach pozwala na export zawartości całej bazy pomiarowej. Podział jest jedynie na pomiary manualne oraz automatyczne. Export danych pomiarowych odbywa się do dokumentu SQL. Lub do XML.



Rys 4.11 Edycja danych w miejscu. [źródło: opracowanie własne]

4.8. Moduł security.

Jest to moduł odpowiedzialny za zabezpieczenie dostępu do systemu osobom niepożądanym. Podczas uruchomienia systemu lub wygaśnięciu sesji, użytkownik jest automatycznie kierowany do formularza logowania. Tam po podaniu hasła oraz loginu jest sprawdzana poprawność podanych danych oraz autentykacja. W razie braku poprawności danych system powróci do formularza i wyświetli stosowny komunikat. Autentykacja polega na utworzeniu w sesji identyfikatora użytkownika oraz nadania mu prawa przeglądania modułów systemu. Konfiguracja dostępu polega na umieszczeniu

pliku security.yml, w katalogu config, w którym określa się czy moduł ma być dostępny tylko po autentykacji.

Listing 4.1 Nadanie modułowi zabezpieczeń

```
all:
  is_secure: on
```

Jest to najprostsza forma określenia dostępu. W konfiguracji można określić bardziej złożone prawa dostępu. Występuje możliwość określenia dostępu dla rożnych typów użytkowników jak i pojedynczego. Można również ograniczać możności co do niektórych akcji modułu. Przykład na listingu 4.2

Listing 4.2 Przykład nadania praw dla wykonywania różnych akcji

```
read:
  is_secure:   off       # Każdy użytkownik może wykonać tą akcje

update:
  is_secure:   on        # Update tylko dla zalogowanych

delete:
  is_secure:   on        # Tylko dla zalogowanych
  credentials: admin     # Tylko z prawami admina

all:
  is_secure:  off        # defaultowa dla pozostałych akcji.
```

W naszym przypadku nie ma potrzeby tworzenia skomplikowanych zestawów praw dostępu, gdyż wymagania systemu nie przewidują tak rozwiniętych zabezpieczeń.

4.9. Obsługa błędów.

Żaden dobry system nie może obejść się bez stworzenia dla niego obsługi błędów. W wymaganiach niefunkcjonalnych zostały przedstawione trzy rodzaje błędów, oraz sposoby jak system ma sobie z nimi radzić.

Błędy krytyczne

Czyli błędy powstałe podczas braku zasobów lub braku dostępu do nich. Może to dotyczyć uruchomienia nieistniejącego skryptu lub próby odczytania z bazy

nieistniejących danych. Takie błędy obsługiwane są przez przekierowanie użytkownika na standardową stronę błędu braku zasobu określoną w konfiguracji – listing 4.3

Listing 4.3 Użycie przekierowania na stronę błędu

```
      $c = new Criteria();
      $this->device = CzujnikPeer::doSelect($c);
      $this->forward404unless(!$this->device);
```

W konfiguracji znajduje się wpis, która strona ma być wyświetlona jako strona błędu 404 oraz w jakim module ona się znajduje:

Listing 4.4 Fragment konfiguracji aplikacji zawartej w pliku settings.yml

```
#all:
#  .actions:
#    default_module:          default   # Default module and action to
be called when
#    default_action:          index     # A routing rule doesn't set it
#
    error_404_module:         errors   # To be called when a 404 error
is raised
    error_404_action:         error404  # Or when the requested URL
doesn't match any route
```

W naszym przypadku jest to moduł errors oraz akcja error404.

Błędy operacyjne

Są to błędy powstające przy nieudanej próbie zapisu do bazy danych lub próbie wykonania operacji dozwolonej tylko pewnej grupie użytkowników. Są one wywoływane również podczas nieudanych prób autentykacji.

Wszystkie te błędy obsługiwane są przez sesyjny mechanizm informacji. Jest to mechanizm, który podczas wystąpienia błędu tworzy krótką informację o powstałym błędzie i umieszcza ją w sesji. Bez względu na to czy dana akcja, która wywołała błąd przekierowuje użytkownika na inna stronę czy też powoduje przeładowanie strony jest ona wychwytywana przez skrypt jak na listingu 4.6. Skrypt wyświetla tą informacje oraz kasuje ją z sesji po wyświetleniu. Rozwiązanie to jest bardzo wygodną i prosta

formą powiadamiania użytkownika o błędach oraz akcjach systemu. Zmieniając jedynie wygląd komunikatu użyłem tego rozwiązania jako formy podawania informacji na temat pracy systemu np. informacji o pomyślnym zakończeniu akcji. Listing 4.5 i 4.6

Listing 4.5 Stworzenie informacji o błędzie

```
$this->setFlash('error', 'Zapis do bazy nie powiódł się');
$this->setFlash('info', 'Operacja powiodła się);
```

Listing 4.6 Skrypt przechwytujący błąd

```
<?php if ($sf_user->hasFlash('error')): ?>
 <div class=”error”> <?php echo $sf_user->getFlash('error') ?> </div>
<?php endif; ?>

<?php if ($sf_user->hasFlash('info')): ?>
<div class=”info”>   <?php echo $sf_user->getFlash('info') ?> </div>
<?php endif; ?>
```

Błędy danych wejściowych

Walidacja jest to mechanizm sprawdzania poprawności danych wejściowych, który informuję nas o niezgodnościach wprowadzonych danych z formatem jaki powinien być wymagany. Błędy danych wejściowych obsługiwane są przez system walidacji udostępniony przez symfony. Symfony dostarcza grupę walidatorów standardowych. Należą do nich:

- sfStringValidator
- sfNumberValidator
- sfEmailValidator
- sfUrlValidator
- sfRegexValidator
- sfCompareValidator
- sfPropelUniqueValidator
- sfFileValidator
- sfCallbackValidator

By wymusić walidacje formularza, który umieściliśmy na naszej stronie musimy stworzyć plik yml w katalogu validate o takiej samej nazwie jak akcja, którą walidujemy. Czyli jeśli chcemy walidować formularz znajdujący się w akcji login (opisanej w pliku action.class.php modułu security) oraz na szablonie loginSucces.php,

plik walidacyjny będzie miał nazwę login.yml. Załóżmy, że walidowane pola to login i password. Plik walidacji dla takiego formularza może wyglądać tak.

Listing 4.7 Plik walidacji dla formularza logowania

```
methods:
  post:               [login, password] # wartości pól które będą
walidowane

fillin:
  activate:           on    #opcja pozwalająca na ponowne wypełnienie
wpisanymi danymi formularza po wystąpieniu błędu

names:
  login:
    required:         true
    required_msg:     login jest wymagany
    validators:       [nicknameValidator] # dodanie dodatkowego
walidatora

  password:
    required:         true
    required_msg:     hasło jest wymagane

nicknameValidator:
    class:            sfStringValidator #walidator łańcuchów znakowych
    param:
      min:            3
      min_error:      login nie krótszy niż 3 znaki
      max:            50
      max_error:      login nie dłuższy niz 50 znaków
```

Walidacja przedstawiona powyżej wymusza wpisania pewnych wartości w oba pola (login i hasło). Dodatkowo pole login ma pewne ograniczenia co do długości. Wyświetlanie informacji o błędach polega na umieszczeniu krótkiego fragmentu kodu nad polem przeznaczonym do walidacji.

Listing 4.8 Wywołanie ewentualnego błędu walidacji

```
<?php echo form_error('login') ?>
<?php echo form_error('password') ?>
```

Działanie z systemem wymusza czasem walidowanie danych pochodzących z dynamicznie tworzonych formularzy. Nie da się jednoznacznie określić ilości np. pól tekstowych oraz narzucić im stałej nazwy. Dzieje się tak na przykład podczas edytowania pomiarów za pomocą wykorzystującego AJAX mechanizmu edycji w miejscu. W takich przypadkach walidację trzeba przeprowadzić manualnie tworząc kod odpowiedzialny za to. Poniżej przedstawiono fragment kodu walidującego edycję pomiaru.

Listing 4.9 Walidacja danych z dynamicznych formularzy

```
public function executeSavedate()
    {
        $id = $this->getRequestParameter('id');
        $c = new Criteria();
        $pomiar = PomiarPeer::retrieveByPk($id);
        if (! $pomiar)
        return $this->raiseEipError( "Nie znaleziono Obiektu o takim
ID" );
if(MyTool->isNumericData($pomiar)){
        $pomiar->setValue( $this->getRequestParameter('value'));
        $pomiar->save();
        return $this->renderText( $this-
>getRequestParameter('value'));
}else{
$this->raiseEipError( "Zły format danych" );

}
    }
    private function raiseEipError ($message)
    {
        //$this->logMessage($message, 'err');
        return $this->renderText( "ERROR:  $message" );
    }
```

Po wykonaniu operacji zapisu sprawdzane jest czy obiekt istnieje w bazie. Następnie sprawdzana jest poprawność wprowadzonej danej. W razie wystąpienia błędów jest wysyłany odpowiedni komunikat.

W założeniach systemu znajduję się możliwość przechowywania pomiarów z urządzeń automatycznych. System wpisuje je do bazy danych po odczytaniu wysłanego do niego pliku xml. Wysyłanie dokumentów do systemu również podlega walidacji. To tego typu działań wykorzystany mechanizm identyczny jak podczas walidacji danych

wejściowych podczas logowania. Poniższy kod przedstawia fragment pliku walidacyjnego odpowiadającego za przesłanie dokumentu xml.

Listing 4.10 Walidacja przesyłania dokumentu

```
myFileValidator:
  class:              sfFileValidator
  param:
    mime_types:
      - 'data/xml
    mime_types_error: Tylko pliki XML
    max_size:         5120000
    max_size_error:   Maxymalny rozmiar to 5MB
```

Powyższy przykład pozwala wysłać do systemu jedynie pliki typu xml oraz ogranicza ich rozmiar do 5 MB.

Bardzo ważnym elementem aplikacji jest walidacja pomiarów. Nie może się zdarzyć sytuacja, w której do pola z pomiarem temperatury wpiszemy wartość np. 100 ponieważ wiadomo, że na ziemi nie występuje taka temperatura powietrza. Formularze do dodawania pomiarów generowane są dynamicznie więc najlepszym rozwiązaniem walidacji takich formularzy jest walidacja po stronie klienta. Mechanizm zastosowany w systemie korzysta z walidacji za pomocą javascript. Jest to najbardziej optymalne rozwiązanie dla dynamicznych formularzy zarówno z powodów wydajnościowych jak i implementacyjnych. Mechanizm ma swój początek na poziomie wprowadzania urządzeń pomiarowych. Tam określa minimalną i maksymalną wartość pomiaru Rys 4.12

Rys 4.12 Dodawanie czujników wraz zakresem danych [źródło: opracowanie własne]

Następnie podczas tworzenia formularza do dodawania pomiarów generowany jest automatycznie kod javascript odpowiedzialny za walidację każdego pola tekstowego w formularzu. Listing 4.12

Listing 4.12 Kod odpowiedzialny za walidacje pól formularza

```
<script type="text/javascript">
function glovalValidate(){

    checkItam("1_1_1",3);
    checkItam("1_1_2",3);
    checkItam("1_1_3",3);
    .
    .
    .
}

// funkcjia walidująca
function checkItam(id,type){
    var error=false;
    var msg = "";

    //walidacjia liczby
    if(document.getElementById(id).value != ""){
        if(validateNumber(document.getElementById(id).value) ==
false){
            error=true;
            msg+="zły format \n";
        }

        if(document.getElementById(id).value>maxRange(type) ||
document.getElementById(id).value<minRange(type) ){
           msg += "przekroczono zakres danych:"
+document.getElementById(id).value+" zakres: <"+minRange(type)+";"
+maxRange(type)+">\n"
           error=true;
        }
    }

    if (!brakuje_danych)
    document.forms[0].submit();
  else
    alert ("Błąd:\n" + msg);

}
<script>
```

Przeglądarka uruchamia globalną funkcje walidującą przy próbie zapisu danych, która sprawdza poprawność wprowadzanych danych. W razie wystąpienia błędu zostanie wyświetlona informacja o nim oraz przeglądarka umieści kursor edycji na polu z błędem. Rys 4.13

Rys 4.13. Informacja o błędzie podczas dodawania pomiarów

5. *Podsumowanie*

Celem powyższej pracy było zaprojektowanie oraz stworzenie systemu gromadzącego dane pomiarowe tradycyjne oraz pochodzące z automatycznych systemów pomiarowych. W pracy przedstawiłem zagadnienia dotyczące procesu tworzenia systemu informatycznego, opisałem technologie, które użyłem do jego wykonania oraz przedstawiłem wynik mojej pracy opisując krótko działanie systemu.

Pierwszą cześć mojej pracy stanowi zapoznanie się z technologia PHP. Przybliżono historie powstania oraz ewolucje samego języka jak i frameworka symfony, który wykorzystano do stworzenia systemu. Zamieszczono informacje na temat wzorca projektowego MVC, przedstawiono przykłady implementacji tego wzorca w krótkiej aplikacji oraz porównanie takiego programu ze standardowym programem nie korzystającym ze wzorca. Opisano strukturę projektu symfony. Zwrócono uwagę na opis warstw modelu, kontrolera, oraz widoku. Przytoczono wiele przykładów skrótów oraz ułatwień, które powodują, że programowanie przy użyciu symfony staje się szybsze i wydajniejsze. Przedstawiono sposób integracji projektu symfony z technologiami bazodanowymi oraz AJAX-em. Na koniec pokazano możliwości testowania stworzonego systemu, które udostępnia framework symfony.

Ważnym elementem, który zawarłem było określenie wymagań systemu gromadzenia danych Taj-FUNv0.2. Określono zbiór wymagań funkcjonalnych, dotyczących sposobu działania systemu, wyglądu interfejsu, kwestii bezpieczeństwa oraz wymagań niefunkcjonalnych, które biorą pod uwagę wymagania systemu, sposób walidacji błędów oraz wydajność. Zwrócono tu uwagę na parametry serwera, na którym system działa.

Zaimplementowany system, którego strukturę wraz z przedstawieniem bazy danych oraz opisem poszczególnych modułów pozwala na gromadzenie danych hydrometeorologicznych w zwartej określonej strukturze. System pozwala na szybki dostęp do danych oraz bardzo łatwy sposób uzupełniania go w kolejne pomiary. System zawiera w bazie dwa rodzaje pomiarów. Pomiary tradycyjne wprowadzane są do systemu za pomocą generowanych formularzy. Formularz generowany jest po wyborze stacji oraz zakresu czasowego pomiaru. Pomiary automatyczne wprowadzane są przez parsowanie plików xml pochodzących z dekodera stworzonego przez Marcina Kowala za pomocą algorytmów parsujących zaproponowanych przez Justynę Kowal. System

pozwala na prezentacje danych w postaci tabelek wprost w oknie przeglądarki, ale także daje możliwość przedstawienia ich na wykresach oraz importowania do formatów CSV, XML i PDF. Prezentacja oraz możliwość edycji zintegrowana jest z intuicyjnym systemem wyszukiwania opartym o możliwości AJAX. Proces wyszukiwania polega na przeciągnięciu wyszukiwanych kryteriów do okna głównego wyszukiwarki. Po określeniu warunków wyszukiwania, tworzone jest z nich zapytanie do bazy danych a rezultat wyświetlany jest już w sposób opisany powyżej.

Podsumowując można uznać iż zaimplementowany system spełnia postawione przed nim wymagania.

6. ***Wnioski***

System, który stworzono okazał się w pełni funkcjonalną aplikacją służąca gromadzeniu hydrometeorologicznych danych pomiarowych co za tym idzie cel pracy został osiągnięty. Zaimplementowano system, który pozwala na zgromadzenia w postaci cyfrowej pomiarów z dosyć szerokiego okresu pomiarowego. Dzięki wykorzystaniu nowoczesnych technologii daje możliwości dostępu do tych danych poprzez Internet lub sieć lokalną co może doskonale wspomóc i przyspieszyć badania, które wykorzystują tego typu pomiary. Stworzony system udostępnia następujące funkcje:

- Moduł zarządzania stacjami pomiarowymi, który daje możliwość tworzenia nowych stacji oraz rozbudowy ich o kolejne urządzenia pomiarowe.
- Dwojaki sposób uzupełnia swojej bazy danymi pomiarowymi. Dane do bazy mogą być dodane po wypełnieniu generowanych przez system formularzy lub przez automatyczne parsowanie plików xml.
- Bardzo prosty system wyszukiwania oraz (w razie potrzeby) edycji danych. Wykorzystane technologie AJAX pozwalają na intuicyjny sposób wykonywania tych czynności.
- Moduł prezentacji współpracujący z modułem wyszukiwania. Pozwala na przedstawienie wyszukanych wyników w postaci tabelarycznej lub export tych danych do formatów takich jak xml, csv czy pdf.

Taki sposób działania systemu znacznie przyspiesza dostęp do danych hydrometeorologicznych oraz poszerza sposób ich wykorzystania.

Tworząc wyżej opisany system postawiono sobie również za cel przybliżenie nowoczesnej technologii tworzenia aplikacji internetowych jaka oferuje framework symfony. Przedstawiono tu, że wykorzystanie tego narzędzie może w znaczny sposób usprawnić i ułatwić tworzenie takich systemów:

- Opisano mechanizmy działania helperów znacznie przyspieszających prace z kodem.
- Generatory, które wspomagają i przyspieszają projektowanie oraz implementacje.
- Proces testowania, który szybko i w łatwy sposób pozwala wykryć błędy implementacyjne.

- Pokazano wykorzystanie wzorca MVC, który doskonale wpływa na przejrzystość oraz funkcjonalność aplikacji.

Powyższe przykłady oraz opisy zawarte w pracy potwierdzają tezę, iż symfony jest doskonałym narzędziem dla programistów aplikacji internetowych. Wypada również podkreślić, że symfony udostępniane jest na licencji Open Source, wiec jest całkowicie darmowe.

*7. **Bibliografia***

1. „The Definitive Guide to Symfony” Fabien Potencier, François Zaninotto
2. „PHP Solution” Software Developer Journal nr 27
3. „AJAX dla twórców aplikacji internetowych” Kris Hadlock , 2007 Helion
4. „PHP5 w praktyce” Elliott White, Jonathan D. Eisenhamer , 2007
5. „Wzorce projektowe. Analiza kodu sposobem na ich poznanie” Allen Holub 2007
6. „Microsoft Official Course Documentation „2663A: Programming with XML in the Microsoft .NET Framework”, 2002 Microsoft Corporation
7. „PHP I MySQL Tworzenie stron internetowych Vademecum profesjionalisty. Wydanie trzecie” Luke Helling, Laura Thompsone, 2005
8. Justyna Kowal „Import i Export danych hydrometeorologicznych z wykorzystaniem formatu XML oraz formatu SAX”, Praca magisterska, Politechnika Krakowska 2008 r.
9. Kowal Marcin „RC10 decoder- system dekodowania i gromadzenia pomiarów hydrometeorologicznych”, Praca magisterska , Politechnika Krakowska 2008

8. *Spis Listing'ów*

Listing 2.5 Płaska aplikacja 13
Listing 2.6 model.php – fragment kodu prezentujący zawartość pliku model 14
Listing 2.7 Kontroler.php – fragment kodu kontrolera 15
Listing 2.8 viev.php – kod przedstawia widok 15
Listing 2.9 Struktura projektu stworzonego w symfony 18
Listing 2.10 Struktura pojedynczej aplikacji 19
Listing 2.11 Struktura modułu 20
Listing 2.12 Typowy Front Controller utworzony przez symfony 22
Listing 2.13 plik action.calss.php 23
Listing 2.14 Wywołanie akcji w module 23
Listing 2.15 Kilka akcji w action.class.php 23
Listing 2.16 Przykładowa akcja 24
Listing 2.17 Przechwytywanie danych z formularza 24
Listnig 2.18 Wysyłanie do widoku 25
Listing 2.19 Zakończenie akcji 25
Listing 2.20 Zwrócenie widoku błędu 26
Listing 2.21 Zwracanie własnego widoku 26
Listing 2.22 Zakończenie bez widoku 26
Listing 2.23 Sposób wymuszenia przejścia do innej akcji 26
Listing 2.25 Przekierowanie na stronę błędu; 27
Listing 2.26 Przechowywanie atrybutów w sesji 27
Listing 2.27 Zakończenie sesji 28
Listing 2.28 Pomocnik (helper) input_tag() 29
Listing 2.29 Domyślny wygląd pliku layout.php 31
Listing 2.30 Ładowanie partiala 32
Listing 2.31 Przykładowa klasa komponentu: 32
Listing 2.32 Widok dla komponentu 33
Listing 2.33 Wywołanie komponentu 33
Listing 2.34 Zmienne w wywołaniu komponentu 33
Listing 2.35 Schemat małej bazy danych składającej się z dwóch tabel w oparciu o standard yml 34
Listing 2.36 Schmat oparty o standard xml 35
Listing 2.37 Przykład dostępu do rekordu 36
Listing 2.38 Pozyskiwanie obiektów modelu 36
Listing 2.39 Działanie na obiektach modelu 36
Listing 2.40 Przykładowy adres URL 37
Listing 2.41 Przyjazny adres URL 38
Listing 2.42 Tłumaczenie adresów przez system rootingu 39
Listing 2.43 Przykłady pomocników (helperów) używanych do nawigacji 40
Listing 2.44 Komenda wywołująca generowanie modułu 41
Listing 2.46 Spis elementów powstałych podczas scoffoldingu 42
Listing 2.47 budowanie modułu administracji dla danej tabeli 43
Listing 2.48 Plik konfiguracyjny dla przykładowego modułu 44
Listing 2.49 Zbiór testow lime 46
Listing 2.50 Wynik testu 47
Listing 2.51 Test funkcjonalny dla modułu 48
Listing 2.52 Wynik testu funkcjonalnego 48
Listing 2.53 Ttworzenie projektu symfony 49
Listing 2.54 Struktura projektu symfony 49

Usunięto

Listing 2.55 Tworzenie nowej aplikacji w projekcie ... 49
Listing 2.56 przykładowa konfiguracja wirtualnego hosta dla aplikacji Symfony ... 50
Listing 2.57 inny sposób na udostępnienie aplikacji ... 50
Listing 2.62 Plik kontrolera ... 51
Listing 2.63 Plik widoku ... 51
Listing 2.64 Plik widoku wspólny dla całej aplikacji ... 52
Listing 4.1 Nadanie modułowi zabezpieczeń ... 69
Listing 4.2 Przykład nadania praw dla wykonywania różnych akcji ... 69
Listing 4.3 Użycie przekierowania na stronę błędu ... 70
Listing 4.4 Fragment konfiguracji aplikacji zawartej w pliku settings.yml ... 70
Listing 4.5 Stworzenie informacji o błędzie ... 71
Listing 4.6 Skrypt przechwytujący błąd ... 71
Listing 4.7 Plik walidacji dla formularza logowania ... 72
Listing 4.8 Wywołanie ewentualnego błędu walidacji ... 72
Listing 4.9 Walidacja danych z dynamicznych formularzy ... 73
Listing 4.10 Walidacja przesyłania dokumentu ... 74
Listing 4.12 Kod odpowiedzialny za walidacje pól formularza ... 75

Usunięto

9. *Spis rysunków*

Rys. 2.1 Schemat ładowania wyglądu modułu do głównego layaut'u). 30
Rys 2.2 Rozróżnienie akcji dla modułu i komponentu. 32
Rys 4.1 Struktura bazy danych. 59
Rys 4.2 Tworzenie urządzenia pomiarowego. 61
Rys 4.3 Tworzenie jednostek dla urządzeń pomiarowych. 62
Rys 4.4 Uzupełnianie stacji urządzeniami pomiarowymi. 63
Rys 4.5 Dodawanie stacji pomiarowych. 63
Rys 4.8 Uzupełnianie formularza z pomiarami. 65
Rys 4.9 Import pomiarów automatycznych. 65
Rys 4.10 Okno tworzenia kryteriów wyszukiwania. 67
Rys 4.11 Edycja danych w miejscu. 68
Rys 4.12 Dodawanie czujników wraz zakresem danych 74
Rys 4.13. Informacja o błędzie podczas dodawania pomiarów 76