Урок 3

Шаблонизаторы

Понятие шаблонизаторов. Знакомство с Twig. Реализации функционала шаблонизатора. Исключения в PHP.

В курсе «PHP. Уровень 1» применялся шаблонизатор, который мы написали самостоятельно. Он выполнял базовый функционал подстановки значений в заранее подготовленные блоки шаблонов и мог работать с простыми циклами. Однако зачастую требуется более мощный функционал при меньших трудозатратах по подготовке. Поэтому мы рассмотрим готовый шаблонизатор, который легко применить в нашем движке интернет-магазина.

Все шаблонизаторы похожи и различаются только специфическими деталями. Главная задача шаблонизаторов – разделить бизнес-логику приложения и вывод данных на страницу, позволив разработчикам и дизайнерам работать одновременно.

Многие PHP-фреймворки, включая **Zend Framework**, **Symfony**, **Yii2** и другие, по-своему реализуют разделение бизнес-логики и вывода данных. Если вы не любите фреймворки или ваш проект слишком мал для их использования, можно воспользоваться отдельной системой построения шаблонов. Мы рассмотрим шаблонизатор **Twig**.

Установка Twig

Самый простой и быстрый способ установить Twig – скачать компонент с GitHub. Архив необходимо распаковать и переместить каталог **lib** в папку с нашим проектом.

Основы

Разберемся с принципом работы шаблонизаторов. Обычное PHP-приложение состоит из целого набора страниц, которые включают в себя как статический HTML-код (меню, списки, изображения), так и динамический контент (вывод данных из БД, xml-файлы, сервисы). С помощью **Twig** мы можем разделить данные процессы, создавая шаблоны со специальными маркерами, вместо которых впоследствии будет вставляться динамический контент.

Значения для данных маркеров формируются в основном PHP-скрипте, и там же происходит общение с базой данных, xml-парсинг и другие операции. Таким образом, страница будет строиться на основе двух источников: шаблона со специальными вставками и PHP-скриптов, где хранится основной функционал. Это дает возможность PHP-разработчикам и дизайнерам одновременно работать над страницами.

Чтобы посмотреть, как работает **Twig**, разберем простой пример:

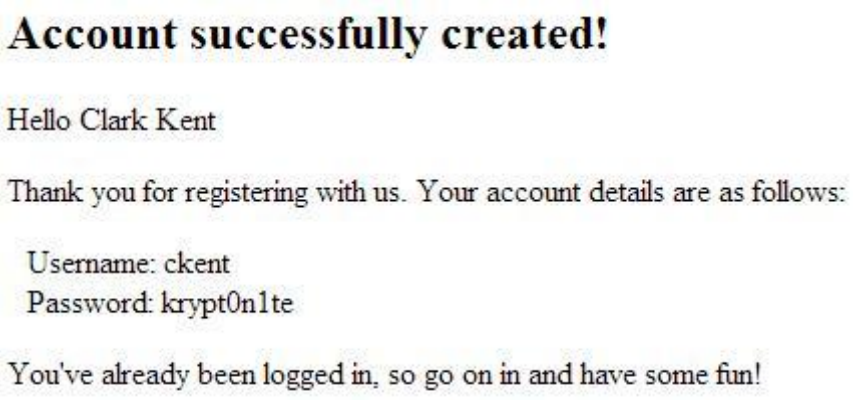
```
<html>
  <head></head>
  <body>
        <h2>Account successfully created!</h2>
        <p>Hello {{ name }}</p>
         <p>Thank you for registering with us. Your account details are as
follows: </p>
        <p style="margin-left: 10px">
        Username: {{ username }} <br/>
        Password: {{ password }}
        </p>
        <p>You've already been logged in, so go on in and have some fun!</p>
  </body>
</html>
```

Обратите внимание на то, что все маркеры, представляющие собой переменные, помещены в двойные фигурные скобки. Подобная запись подскажет **Twig**, где и как вставлять данные. В предыдущей версии нашего движка использовался такой же синтаксис.

Затем создадим основной скрипт, где будут формироваться переменные и данные:

```
<?php
// Подгружаем и активируем автозагрузчик Twig-а
require_once 'Twig/Autoloader.php';
Twig_Autoloader::register();
try {
// Указывает, где хранятся шаблоны
  $loader = new Twig_Loader_Filesystem('templates');
// Инициализируем Twig
  $twig = new Twig_Environment($loader);
// Подгружаем шаблон
  $template = $twig->loadTemplate('thanks.tmpl');
// Передаем в шаблон переменные и значения
// Выводим сформированное содержание
  echo $template->render(array(
    'name' => 'Clark Kent',
    'username' => 'ckent',
    'password' => 'krypt0n1te',
  ));
 } catch (Exception $e) {
  die ('ERROR: ' . $e->getMessage());
}
?>
```

В результате, если открыть данную страницу в браузере, увидим следующее:



Таким образом, для использования **Twig** нужно выполнить следующие шаги:

1. **Инициализировать автозагрузчик Twig** – чтобы классы шаблонизатора подгружались автоматически;
2. **Инициализировать загрузчик шаблонов** – в нашем случае это **Twig_Loader_FileSystem**. В качестве аргумента передаем путь к каталогу с шаблонами;
3. **Создать объект самого Twig и передать ему уже сконфигурированные настройки**;
4. **Подгрузить нужный шаблон с помощью метода loadTemplate**, передав в него название используемого шаблона. В качестве результата метод вернет экземпляр шаблона;
5. **Сформировать массив вида «ключ-значение»**, где ключи – это названия переменных, а значения – данные, выводимые в шаблоне. Затем этот массив нужно передать в метод **render()**, который совместит шаблон с переданными данными и вернет сгенерированный результат.

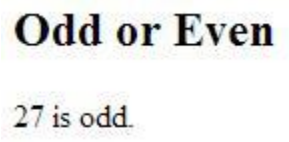
Условные операторы в Twig

Twig также предоставляет возможность создавать условные выражения **‘if-else-endif’**. Пример:

```
<html>
  <head></head>
  <body>
      <h2>Odd or Even</h2>
          {% if div == 0 %}
            {{ num }} is even.
          {% else %}
            {{ num }} is odd.
          {% endif % }
  </body>
</html>
```

В зависимости от числа, которое генерируется в основном PHP-скрипте, шаблон отобразит одно из двух сообщений. Вот скрипт, где генерируется число от 0 до 30 и проверяется на четность:

```
<?php
      include 'Twig/Autoloader.php';
Twig_Autoloader::register();
try {
  $loader = new Twig_Loader_Filesystem('templates');
  $twig = new Twig_Environment($loader);
  $template = $twig->loadTemplate('numbers.tmpl');
// Генерируем случайное число
// Проверяем его на четность
  $num = rand (0,30);
  $div = ($num % 2);
  echo $template->render(array (
    'num' => $num,
    'div' => $div
  ));
} catch (Exception $e) {
  die ('ERROR: ' . $e->getMessage());
}
?>
```

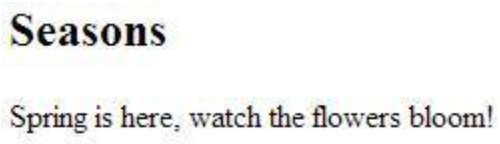


Можно сделать многоуровневые проверки **‘if-elseif-else-endif’**. Пример:

```
<html>
      <head></head>
        <body>
          <h2>Seasons</h2>
          {% if month > 0 and month <= 3 %}
            Spring is here, watch the flowers bloom!
          {% elseif month > 3 and month <= 6 %}
            Summer is here, time to hit the beach!
          {% elseif month > 6 and month <= 9 %}
            Autumn is here, watch the leaves slowly fall!
          {% elseif month > 9 and month <= 12 %}
            Winter is here, time to hit the slopes!
          {% endif %}
        </body>
</html>
```

Скрипт, где генерируется номер месяца и передается в шаблон:

```
<?php
include 'Twig/Autoloader.php';
Twig_Autoloader::register();
try {
  $loader = new Twig_Loader_Filesystem('templates');
  $twig = new Twig_Environment($loader);
   $template = $twig->loadTemplate('seasons.tmpl');
// Получаем номер месяца
  $month = date('m', mktime());
  echo $template->render(array (
    'month' => $month
  ));
} catch (Exception $e) {
  die ('ERROR: ' . $e->getMessage());
}
?>
```



Циклы в Twig

Twig также поддерживает цикл **'for'**. Он удобен, если необходимо пройтись по массиву. Пример:

```
<html>
  <head></head>
  <body>
    <h2>Shopping list</h2>
    <ul>
      {% for item in items %}
        <li>{{ item }}</li>
      {% endfor %}
    </ul>
  </body>
</html>
```

В данном примере – простой неассоциативный массив. На каждой итерации будем получать по одному элементу и выводить его в элементе списка. Скрипт:

```php
<?php
// Формируем массив
$items = array(
  'eye of newt',
  'wing of bat',
  'leg of frog',
  'hair of beast'
);
 include 'Twig/Autoloader.php';
Twig_Autoloader::register();
try {
  $loader = new Twig_Loader_Filesystem('templates');
  $twig = new Twig_Environment($loader);
  $template = $twig->loadTemplate('list.tmpl');
  echo $template->render(array (
    'items' => $items
  ));
} catch (Exception $e) {
  die ('ERROR: ' . $e->getMessage());
}
?>
```

Чтобы пройтись по ассоциативному массиву, можно обращаться к ключам через «точку». Пример:

```
<?php
// Готовим ассоциативный массив
      $book = array(
        'title'     => 'Harry Potter and the Deathly Hallows',
        'author'    => 'J. K. Rowling',
        'publisher' => 'Scholastic',
        'category'  => 'Children\'s fiction',
        'pages'     => '784'
      );
      include 'Twig/Autoloader.php';
      Twig_Autoloader::register();
      try {
        $loader = new Twig_Loader_Filesystem('templates');
        $twig = new Twig_Environment($loader);
        $template = $twig->loadTemplate('book.tmpl');
        echo $template->render(array (
          'book' => $book
        ));
      } catch (Exception $e) {
        die ('ERROR: ' . $e->getMessage());
      }
      ?>
```

Чтобы достучаться до значений массива в шаблоне, сначала пишем имя переменной, в которой хранится сам массив. Затем ставим точку и вводим название ключа, по которому достаем данные:

```
<html>
        <head>
          <style type="text/css">
            table {
              border-collapse: collapse;
            }
            tr.heading {
              font-weight: bolder;
            }
            td {
              border: 1px solid black;
              padding: 0 0.5em;
            }
          </style>
        </head>
        <body>
          <h2>Book details</h2>
          <table>
            <tr>
              <td><strong>Title</strong></td>
              <td>{{ book.title }}</td>
            </tr>
            <tr>
              <td><strong>Author</strong></td>
              <td>{{ book.author }}</td>
            </tr>
            <tr>
              <td><strong>Publisher</strong></td>
              <td>{{ book.publisher }}</td>
            </tr>
            <tr>
              <td><strong>Pages</strong></td>
              <td>{{ book.pages }}</td>
            </tr>
            <tr>
              <td><strong>Category</strong></td>
              <td>{{ book.category }}</td>
            </tr>
          </table>
        </body>
</html>
```

Book details

Title	Harry Potter and the Deathly Hallows
Author	J. K. Rowling
Publisher	Scholastic
Pages	784
Category	Children's fiction

Такой же подход может быть применен для работы с объектами.

Дамп данных

Циклы пригодятся при выводе данных из БД. Пример:

```
<html>
        <head>
          <style type="text/css">
            table {
              border-collapse: collapse;
            }
            tr.heading {
              font-weight: bolder;
            }
            td {
              border: 1px solid black;
              padding: 0 0.5em;
            }
          </style>
        </head>
        <body>
          <h2>Countries and capitals</h2>
          <table>
            <tr class="heading">
              <td>Country</td>
              <td>Region</td>
              <td>Population</td>
              <td>Capital</td>
              <td>Language</td>
            </tr>
            {% for d in data %}
            <tr>
              <td>{{ d.name|escape }}</td>
              <td>{{ d.region|escape }}</td>
              <td>{{ d.population|escape }}</td>
              <td>{{ d.capital|escape }}</td>
              <td>{{ d.language|escape }}</td>
            </tr>
            {% endfor %}
          </table>
        </body>
</html>
```

В следующем фрагменте кода используется PDO-подключение к **MySQL** базе данных **‘world’**. Чтобы попробовать данный пример, нужно сформировать базу самостоятельно:

```
<?php
include 'Twig/Autoloader.php';
Twig_Autoloader::register();
// Подключение к бд
try {
  $dbh = new PDO('mysql:dbname=world;host=localhost', 'root', 'guessme');
} catch (PDOException $e) {
  echo "Error: Could not connect. " . $e->getMessage();
}
// Установка error-режима
$dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
// Выполняем запрос
try {
// Формируем SELECT-запрос
// В результате каждая строка таблицы будет объектом
  $sql = "SELECT country.Code AS code, country.Name AS name, country.Region AS region, country.Population AS population, countrylanguage.Language AS language, ity.Name AS capital FROM country, city, countrylanguage WHERE country.Code = city.CountryCode AND country.Capital = city.ID AND country.Code = countrylanguage.CountryCode AND countrylanguage.IsOfficial = 'T' ORDER BY population DESC LIMIT 0,20";
  $sth = $dbh->query($sql);
  while ($row = $sth->fetchObject()) {
    $data[] = $row;
  }
// Закрываем соединение
  unset($dbh);
   $loader = new Twig_Loader_Filesystem('templates');
  $twig = new Twig_Environment($loader);
  $template = $twig->loadTemplate('countries2.tmpl');
  echo $template->render(array (
    'data' => $data
  ));
} catch (Exception $e) {
  die ('ERROR: ' . $e->getMessage());
}
?>
```

Здесь используется метод **fetchObject()**, который вернет строки из таблицы в виде объектов. Названия полей будут соответствовать наименованиям колонок. Затем эти объекты следует поместить в массив и передать его в шаблон. В шаблоне используем цикл и выводим данные.

В данном примере также применяется встроенный в **Twig** фильтр **`escape`**. По умолчанию он пользуется функцией **htmlspecialchars()** для фильтрования данных. Это неплохая защита от **XSS**-атак.

Подгрузка шаблонов

В **Twig** есть команда **`include`**, которая позволяет подключать содержание других шаблонов. Это может пригодиться, чтобы прикрепить к файлам меню, заголовок или подвал.

Представим, что данный код – это главный шаблон:

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="main.css" />
  </head>
  <body>
    <div id="page">
      <div id="header">
      {% include 'primary.tmpl' %}
      </div>
      <div id="left">
        {% include 'secondary.tmpl' %}
      </div>
      <div id="right">
       This is the main page content. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
      </div>
      <div id="footer">
        {% include 'footer.tmpl' %}
      </div>
    </div>
  </body>
</html>
```

Все секции этой страницы находятся в отдельных файлах и подключаются сюда с помощью команды `**include**`. Посмотрим, как выглядят подключаемые файлы:

```
<!-- begin: primary.tmpl -->
    <table>
      <tr>
        {% for item in nav.primary %}
        <td><a href="{{ item.url }}">{{ item.name|upper }}</a></td>
        {% endfor %}
      </tr>
    </table>
<!-- end: primary.tmpl -->
```

```
<!-- begin: secondary.tmpl -->
   <ul>
     {% for item in nav.secondary %}
     <li><a href="{{ item.url }}">{{ item.name }}</a></li>
     {% endfor %}
   </ul>
<!-- end: secondary.tmpl -->
```

```
<!-- begin: footer.tmpl -->
 <div style="align:center">
  This page licensed under a Creative Commons License. Last updated on: {{
updated }}.
 </div>
<!-- end: footer.tmpl -->
```

Вот главный PHP-скрипт:

```
<?php
// формируем массив
$nav = array(
  'primary' => array(
    array('name' => 'Clothes', 'url' => '/clothes'),
    array('name' => 'Shoes and Accessories', 'url' => '/accessories'),
    array('name' => 'Toys and Gadgets', 'url' => '/toys'),
    array('name' => 'Books and Movies', 'url' => '/media'),
  ),
  'secondary' => array(
    array('name' => 'By Price', 'url' => '/selector/v328ebs'),
    array('name' => 'By Brand', 'url' => '/selector/gf843k2b'),
    array('name' => 'By Interest', 'url' => '/selector/t31h393'),
    array('name' => 'By Recommendation', 'url' => '/selector/gf942hb')
  )
);
include 'Twig/Autoloader.php';
Twig_Autoloader::register();
try {
  $loader = new Twig_Loader_Filesystem('templates');
  $twig = new Twig_Environment($loader);
  $template = $twig->loadTemplate('shop.tmpl');
  echo $template->render(array (
    'nav' => $nav,
    'updated' => '24 Jan 2011'
  ));
} catch (Exception $e) {
  die ('ERROR: ' . $e->getMessage());
}
?>
```

Обратите внимание, что не нужно загружать все шаблоны функцией **loadTemplate**. Главное – подключить основной шаблон. Каждый мелкий подшаблон загрузится автоматически. Переменные и значения, переданные в главный шаблон, будут доступны во всех подключаемых шаблонах.

CLOTHES SHOES AND ACCESSORIES TOYS AND GADGETS BOOKS AND MOVIES

- By Price
- By Brand
- By Interest
- By Recommendation

This page licensed under a Creative Commons License. Last updated on: 24 Jan 2011.

Фильтрация данных

Рассмотрим возможности, которые предоставляет **Twig** в сфере фильтрации данных.

К примеру, фильтр **‘date’** позволяет формировать дату и время, используя нативные для PHP маркеры. Пример:

```
<html>
  <head></head>
  <body>
  {{ "now"|date("d M Y h:i")  }} <br/>
  {{ "now"|date("d/m/y")  }}
  </body>
</html>
```

Результат:

06 Mar 2013 08:48
06/03/13

Также можно воспользоваться фильтрами `**upper**`, `**lower**`, `**capitalize**`, `**title**` для контроля заглавных и прописных букв:

```
<html>
  <head></head>
  <body>
  {{ "the cow jumped over the moon"|upper  }} <br/>
  {{ "the cow jumped over the moon"|capitalize  }} <br/>
  {{ "the cow jumped over the moon"|title  }} <br/>
  {{ "The Cow jumped over the Moon"|lower  }} <br/>
  </body>
</html>
```

Результат:

THE COW JUMPED OVER THE MOON
The cow jumped over the moon
The Cow Jumped Over The Moon
the cow jumped over the moon

Фильтр `**striptags**` уберет из текста все HTML и XML-элементы:

```
<html>
  <head></head>
  <body>
  {{ "<div>I said \"<b>Go away!</b>\"</div>"|striptags  }} <br/>
  </body>
</html>
```

Результат:

I said "Go away!"

Фильтр `**replace**` позволяет быстро заменять значения в строке:

```
<html>
  <head></head>
  <body>
  {{ "I want a red boat"|replace({"red" : "yellow", "boat" : "sports car"})  }}
<br/>
  </body>
</html>
```

I want a yellow sports car

В Twig есть фильтр, который выполняет действие, противоположное `**escape**` – это `**raw**`. Его следует использовать только для html-кода, который вы считаете на 100% безопасным:

```
<html>
  <head></head>
  <body>
  Escaped output: {{ html|escape }} <br/>
  Raw output: {{ html|raw }} <br/>
  </body>
</html>
```

Если же нужно применить `**escape**` к большому блоку кода, можно воспользоваться синтаксисом `**autoescape**`, передав булево значение **true/false** для активации и дезактивации фильтрации `**escape**`. Пример:

```
<html>
  <head></head>
  <body>
  {% autoescape true %}
  Escaped output: {{ html }} <br/>
  {% endautoescape %}
  {% autoescape false %}
  Raw output: {{ html }}
  {% endautoescape %}
  </body>
</html>
```

Теперь вы знаете о **Twig** достаточно, чтобы использовать условия, циклы и фильтры.

Исключения (Exceptions)

В примерах с подключением библиотек **Twig** встречалась конструкция **try…catch**. Она непосредственно связана с сущностями, крайне важными с точки зрения архитектуры, – исключениями.

Исключения – это специальные условия, которые проявляются или могут быть специально созданы программой (обычно, в случае ошибки). Они указывают, что нечто в процессе отклоняется от предполагаемого хода событий. В большинстве случаев подобные ситуации требуют выполнения специальных инструкций, чтобы предотвратить неконтролируемое завершение процесса.

Исключения могут быть брошены (**thrown**) и пойманы (**caught**). Когда исключение брошено – произошло что-то, выходящее за рамки нормальной работы программы, и требуется выполнить другую функцию. Подхват исключения осуществляется в специальной функции, которая сообщает остальной программе о том, что она готова обработать исключение.

Сформировать обработку исключений в PHP очень просто. Сами исключения являются частью стандартной библиотеки **PHP SPL**, которая входит в ядро PHP, начиная с версии 5, и не требует дополнительных подключений.

Исключения реализуются так же, как и любой другой объект:

```
<?php
      $exception = new Exception();
      throw $exception;
?>
```

У них есть методы, которые можно вызвать, чтобы облегчить формирование реакции на исключение.

Методы исключений

- **getMessage()** – получает сообщение исключения;
- **getCode()** – возвращает числовой код, который представляет исключение;
- **getFile()** – возвращает файл, в котором произошло исключение;
- **getLine()** – возвращает номер строки в файле, где произошло исключение;
- **getTrace()** – возвращает массив **backtrace()** до возникновения исключения;
- **getPrevious()** – возвращает исключение, произошедшее перед текущим, если оно было;
- **getTraceAsString()** – возвращает массив **backtrace()** исключения в виде строки;
- **__toString()** – возвращает все исключение в виде строки. Данную функцию можно переписать.

Исключения хороши тем, что это стандартизованный механизм. Благодаря этому человек, не работавший с вашим кодом, не будет вынужден читать мануал, чтобы понять, как обрабатывать ошибки. Ему достаточно знать, как работают исключения. При этом с исключениями гораздо проще находить источник ошибок, так как всегда есть стек вызовов (**trace**).

Если какое-либо из действий выбрасывает исключение, его можно поймать в месте выполнения кода. Для этого нужно обернуть выполняемый блок в конструкцию **try…catch**:

```php
<?php
class A{
  function myMethod(){
    // do something
    throw new Exception("Exception time!");
  }
}
?>
```

```php
<?php
$my_a = new A();
try{
   $my_a-> myMethod();
}
catch(Exception $e){
   echo $e->getMessage();
}
?>
```

В этом примере мы выбрасываем исключение в методе класса **A**. При работе с ним мы берем его в блок **try…catch**. Если этого не сделать, в случае выброса исключения мы получим ошибку **Unhandled exception** (необработанное исключение). Обычно **IDE** отслеживают возможность выброса исключений и подсказывают, когда нужно обернуть тот или иной блок в **try…catch**.

В наших примерах исключения обрабатываются достаточно грубо – мы выключаем скрипт, но даем понять, что именно и где пошло не так. Обычно исключение логируется в специальное хранилище для последующего изучения причин и их устранения.

Исключения могут наследоваться, как и другие классы. В идеале должен быть базовый класс исключений, от которого наследуются все исключения, бросаемые в коде. Блоков **catch** может быть много, по одному на каждый класс перехватываемых исключений. Таким образом можно создать фильтр исключений, т.е. перехватывать не все, а только избранные типы исключений. Все остальные будут перехвачены стандартным обработчиком PHP.

В PHP 5.5 и более поздних версиях также можно использовать блок **finally** после или вместо **catch**. Код в блоке **finally** всегда будет выполняться после кода в блоках **try** и **catch**, вне зависимости от того, было брошено исключение или нет (перед тем, как продолжится нормальное выполнение кода).

Хорошим тоном при создании сайтов считается полное отсутствие ошибок. Если же ошибки возникают – они должны быть обработаны и представлены пользователю и разработчику в понятном виде.

Практическое задание

1. Создать галерею изображений, состоящую из двух страниц:
    a. Просмотр всех фотографий (уменьшенных копий).
    b. Просмотр конкретной фотографии (изображение оригинального размера).
    c. Все страницы вывода на экран – это twig-шаблоны. Вся логика – на backend.
    d. * Реализовать хранение ссылок и информации по картинкам в БД.
2. * Для примера 6 из сегодняшнего урока реализовать хранение в БД, которое позволит логике **example6.php** работать.

Дополнительные материалы

1. http://ruseller.com/lessons.php?rub=37&id=1652 – Twig в примерах.
2. http://ruseller.com/lessons.php?rub=37&id=1277 – исключения в примерах.
3. https://github.com/fabpot/Twig/ – Twig-исходники.
4. https://habrahabr.ru/post/100137/ – как правильно применять исключения?

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Мэтт Зандстра. PHP. Объекты, шаблоны и методики программирования.
2. http://php.net/manual/ru/language.exceptions.php – исключения (exceptions) в PHP.