



**POLITECHNIKA
RZESZOWSKA**
im. IGNACEGO ŁUKASIEWICZA

Aplikacje Internetowe – dokumentacja :

- System zarządzania artykułami :

 - Opis systemu
 - Opis tworzenia modułu do Zend Framework 2 operującego artykułami

Nazwisko i imię: Sura Piotr Krzysztof

Adres E-Mail: piotr@sura.eu

Numer indeksu: 127645

Grupa: EFCDUAIL7

Prowadzący: dr inż. Tomasz RAK

Spis treści

1. Opis systemu	3
• Opis funkcjonalności systemu	3
• Diagram przypadków użycia	4
• Technologia i środowisko	5
• Repozytorium	5
2. Opis tworzenia modułu do Zend Framework 2 operującego artykułami	6
• Struktura modułu (drzewo projektu)	6
• Konfiguracja autoloadera	7
• Konfiguracja modułu	8
• Włączenie modułu	9
• Trasowanie	9
• Tworzenie kontrolera	10
• Tworzenie widoków	11
• Baza danych	12
• Model	12
• Formularze	14
• Walidacja formularzy	15
• Autoryzacja	16
3. Prezentacja działania modułu	17

Opis systemu

1. Opis funkcjonalności systemu

Aplikacja Article_Finder jest systemem informatycznym wspomagającym zarządzanie artykułami. Pozwala między innymi na dodawanie nowych artykułów, ich edycję, wyszukiwanie oraz przeglądanie. Aplikacja wyposażona jest w edytor tekstowy WYSIWYG.

W systemie występują 3 typy użytkowników, posiadających odmienne uprawnienia:

1. Użytkownik niezarejestrowany

Podstawowa funkcjonalność:

- Przeglądanie najnowszych artykułów

2. Użytkownik zarejestrowany

Dodatkowa funkcjonalność:

- Logowanie
- Wylogowanie
- Zmiana hasła
- Zmiana adresu email
- Tworzenie nowego artykułu
- Edycja artykułu
- Eksportowanie artykułu do pliku PDF
- Drukowanie artykułu
- Przeglądanie artykułów z uwzględnieniem wyboru kategorii

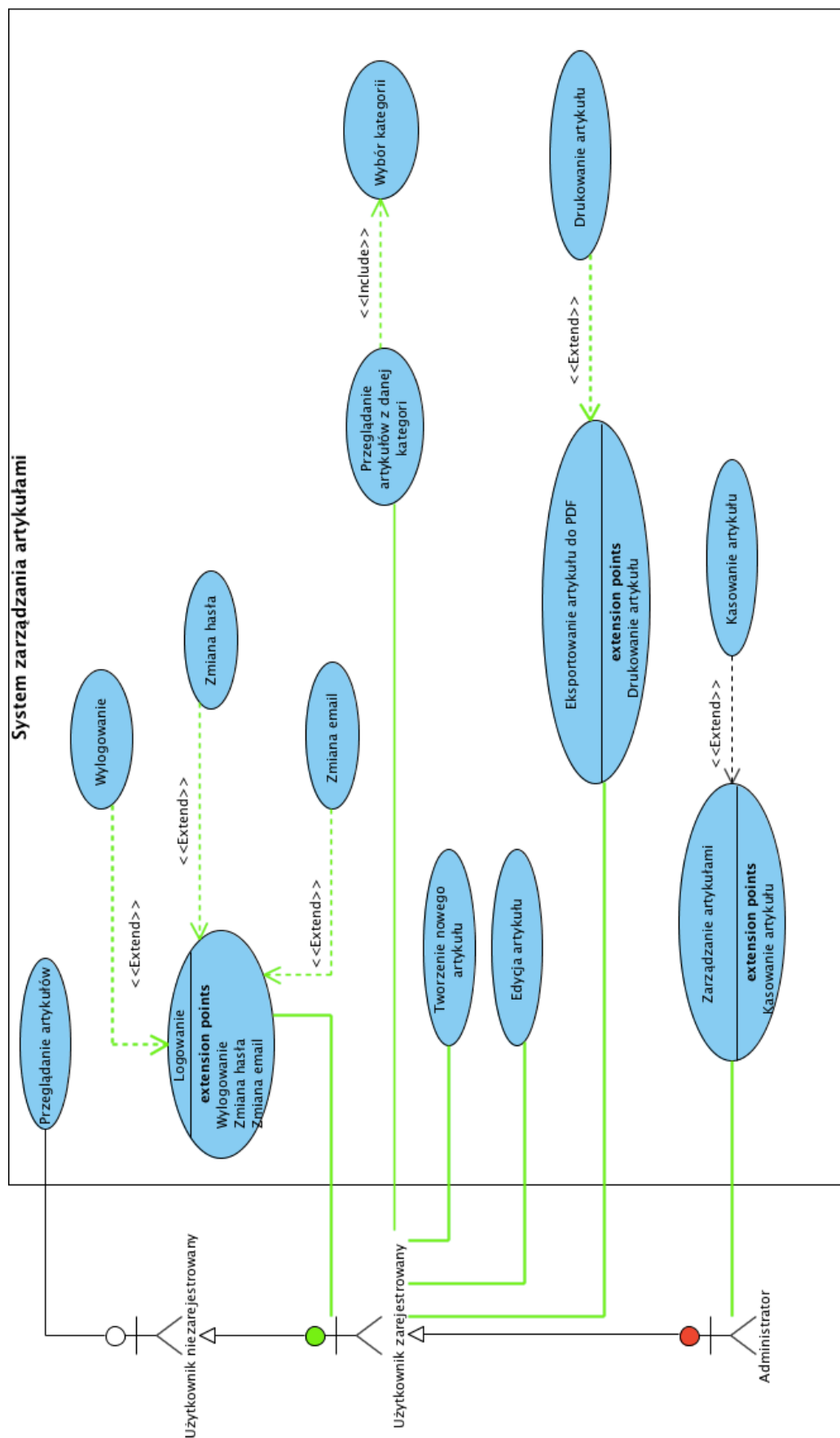
3. Administrator

Dodatkowa funkcjonalność:

- Zarządzanie artykułami (wyświetlanie daty utworzenia/modyfikacji, adresu email autora artykułu, id oraz tytuł artykułu)

Dostępne moduły*: moduł logowania(logowanie, wylogowywanie), moduł zarządzania artykułami(dodawanie nowych, usuwanie oraz edycja istniejących, itp.), , moduł zarządzania autoryzacją, moduł administratora.

2. Diagram przypadków użycia



3. Technologia i środowisko

Technologia: **PHP**

Framework PHP: **Zend Framework 2**

Baza danych: **MySQL**

System operacyjny: **Mac OS X 10.10.5**

Środowisko programistyczne: **NetBeans 8.0.2**

4. Repozytorium

Serwis hostingowy: **GitHub**

Nazwa użytkownika: **PiotrMac**

Nazwa repozytorium: **Article_Finder**

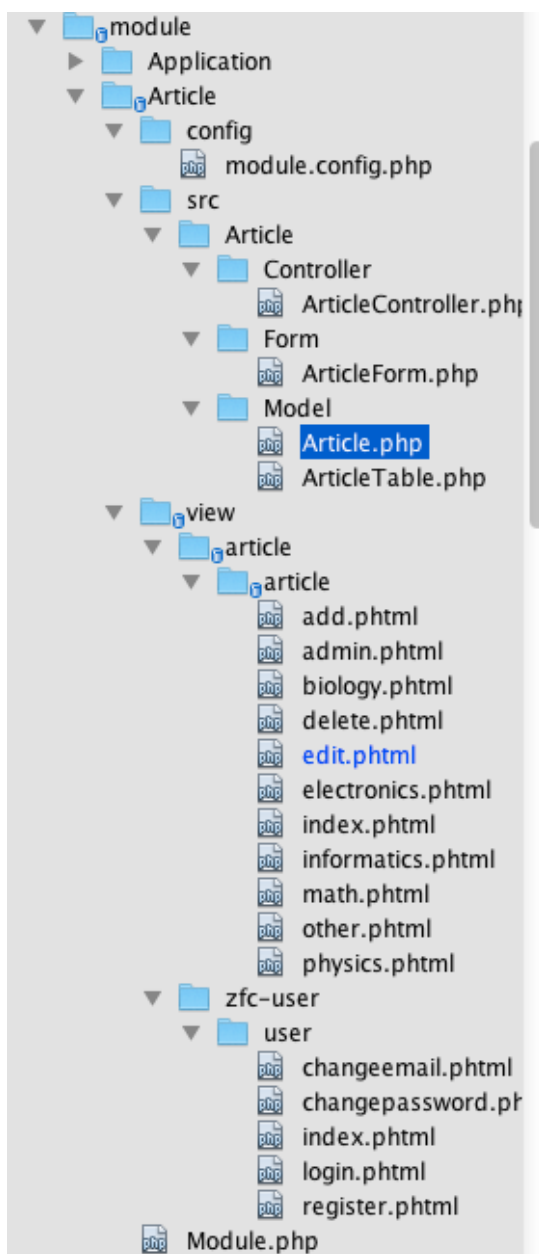
Adres URL repozytorium: **https://github.com/PiotrMac/Article_Finder**

Opis tworzenia modułu do Zend Framework 2 operującego artykułami

1. Struktura modułu(drzewo projektu)

Każdy moduł w Zend Framework 2 musi posiadać specjalny, określony w dokumentacji, układ plików oraz teczek. Główna teczka z modulem (o nazwie takiej jak moduł) zawiera trzy teczki podrzędne dla różnych typów plików:

- config - miejsce na pliki konfiguracyjne modułu,
- src - miejsce na pliki PHP zawierające klasy wykorzystujące przestrzeń nazw modułu,
- view - miejsce na pliki odpowiadające za wygląd modułu.



Rys. 1 Pełna struktura modułu obsługującego artykuły

2. Konfiguracja Autoloadera

Po skonfigurowaniu drzewa modułu, w głównej ścieżce, należy utworzyć plik `Module.php`. Implementuje on dwie metody: `getAutoloaderConfig()` oraz `getConfig()`. Pierwsza odpowiada za automatyczną konfigurację przestrzeni nazw dla każdej klasy zawartej w module. Druga dołącza plik konfiguracyjny modułu (znajdujący się w teczce o nazwie `config`).

W trakcie ładowania modułu, Zend Framework 2 uruchamia `ModuleManager`, który to z kolei automatycznie wywołuje wspomniane wcześniej metody, umożliwiając poprawne załadowanie modułu.

Na Rys. 2 przedstawiono zawartość omawianej klasy. Należy zauważyć wykorzystanie alternatywnej wersji implementacji metody `getAutoloaderConfig()`. W prezentowanym przykładzie funkcjonalność metody została zaimplementowana w pliku `composer.json`

```
1<?php
2namespace Article;
3
4use Article\Model\Article;
5use Article\Model\ArticleTable;
6use Zend\Db\ResultSet\ResultSet;
7use Zend\Db\TableGateway\TableGateway;
8use Zend\ModuleManager\Feature\AutoloaderProviderInterface;
9use Zend\ModuleManager\Feature\ConfigProviderInterface;
10
11class Module implements AutoloaderProviderInterface, ConfigProviderInterface
12{
13    public function getAutoloaderConfig()
14    {
15    }
16
17    public function getConfig()
18    {
19        return include __DIR__ . '/config/module.config.php';
20    }
21}
```

Rys. 2 Fragment pliku `Module.php`

3. Konfiguracja modułu

Aby dokonać konfiguracji modułu należy utworzyć plik `module.config.php` w teczce `config`. Na Rys. 3 zaprezentowano zawartość pliku konfiguracyjnego wykorzystywanego w naszej aplikacji. Jak widać zawiera on definicje controllera oraz menadżera widoków. W późniejszym kroku dodane zostaną deklaracje ścieżek dostępu pozwalających na przemieszczanie się pomiędzy akcjami (stronami) zaimplementowanymi w module(ang. routing) – więcej informacji w rozdziale Trasowanie.

```
return array(
    'controllers' => array(
        'invokables' => array(
            'Article\Controller\Article' => 'Article\Controller\ArticleController',
        ),
    ),
    'router' => array(
        'routes' => array(
            'article' => array(
                'type' => 'segment',
                'options' => array(
                    'route' => '/article[/:action][/:id_artykulu]',
                    'constraints' => array(
                        'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
                        'id_artykulu' => '[0-9]+',
                    ),
                    'defaults' => array(
                        'controller' => 'Article\Controller\Article',
                        'action' => 'index',
                    ),
                ),
            ),
        ),
    ),
    'view_manager' => array(
        'template_path_stack' => array(
            'article' => __DIR__ . '/../view',
        ),
    ),
);
```

Rys. 3 Kompletny plik konfiguracyjny modułu – `module.config.php`

4. Włączenie modułu

Aby ModuleManager mógł załadować dowolny moduł w aplikacji Zend Framework 2 należy dodać jego przestrzeń nazw do listy modułów w głównym pliku konfiguracyjnym aplikacji – application.config.php.

5. Trasowanie

W aplikacji wykorzystującej Zend Framework 2 każda podstrona rozumiana jest jako osobna akcja, należąca do danego kontrolera. Konfiguracji mapowania adresów URL należy dokonać we wspomnianym w rozdziale 3 pliku module.config.php.

Na Rys 4 zaprezentowano kompletne trasowanie dla utworzonego modułu.

Jak można zauważyć, najważniejszą częścią jest lista 'options'. W niej zawarto pole 'route', definiujące możliwe adresy URL. Elementy pomiędzy nawiasami kwadratowymi są ścieżkami opcjonalnymi. Możliwe znaki definiowanych adresów zawarto w liście 'constraints'. W nawiasach kwadratowych definiujemy pojedyncze znaki adresu. Znak * oznacza, że kolejne znaki nie są wymagane. W opisywanym module, akcje mogą zostać wykonane tylko dla wybranego artykułu (parametr id_artykułu, w formie liczby typu int).

```
'router' => array(
    'routes' => array(
        'article' => array(
            'type' => 'segment',
            'options' => array(
                'route' => '/article[:action] [:id_artykułu]',
                'constraints' => array(
                    'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
                    'id_artykułu' => '[0-9]+',
                ),
                'defaults' => array(
                    'controller' => 'Article\Controller\Article',
                    'action' => 'index',
                ),
            ),
        ),
    ),
),
```

Rys. 4 Definicja trasowania dla modułu Article

6. Tworzenie kontrolera

Zanim przystąpimy do utworzenia kontrolera należy zaplanować akcje, które mają zostać obsługane przez kontroler. W przypadku opisywanego modułu zaimplementowane zostały następujące akcje:

- indexAction - wyświetlanie wszystkich artykułów,
- <nazwa_kategorii>Action, wyświetlanie artykułów z podziałem poszczególne na kategorie,
- addAction - tworzenie nowych artykułów,
- editAction - edycja oraz usuwanie artykułów
- adminAction - obsługa panelu administratora.

Kontroler zadeklarowany jest w postaci klasy ArticleController, dziedziczącej po AbstractActionController. Plik ArticleController.php znajduje się w teczce src/Article/Controller.

Każda akcja reprezentowana jest jako metoda publiczna klasy ArticleController.

Na Rys. 5 zaprezentowano fragment abstrakcyjnej klasy kontrolera zawierający definicję metody indexAction.

```
<?php

namespace Article\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;
use Article\Model\Article;
use Article\Form\ArticleForm;

class ArticleController extends AbstractActionController
{
    protected $articleTable;
    //protected $categoryTable;

    public function indexAction()
    {
        // grab the paginator from the ArticleTable
        $paginator = $this->getArticleTable()->fetchAll(true);
        // set the current page to what has been passed in query string, or to 1 if none set
        $paginator->setCurrentPageNumber((int) $this->params()->fromQuery('page', 1));
        // set the number of items per page to 1
        $paginator->setItemCountPerPage(1);

        return new ViewModel(array(
            'paginator' => $paginator
        ));
    }
}
```

Rys. 5 Fragment pliku ArticleController.php

7. Tworzenie widoków

Widoki odpowiadają za stronę wizualną tworzonego modułu. W opisywanym module wykorzystano pięć widoków dla głównych akcji oraz po jednym widoku dla każdej kategorii artykułów. Każdy widok zapisany jest w postaci pliku ze skryptem w formacie nazwa_widoku.phtml oraz znajduje się w teczce Article/view/article/article.

Na Rys. 6 zaprezentowano zawartość pliku realizującego widok dla akcji add. Jak widać tworzony jest formularz dodawania nowych artykułów (więcej w rozdziale Model). Warto również zauważyć sposób implementacji edytora WYSIWYG (w tym przypadku jest to ckeditor). Na początku pliku ładujemy skrypt edytora, a następnie podmieniamy pole tekstowe „tekst_ckeditor” formularza na edytor WYSIWYG.

```
<?php
namespace Article;

$title = 'Dodaj nowy artykuł';
$this->headTitle($title);
?>

<script src="../../ckeditor/ckeditor.js"></script>
<h1><?php echo $this->escapeHtml($title); ?></h1>

<?php

$form->setAttribute('action', $this->url('article', array('action' => 'add')));
$form->prepare();

echo $this->form()->openTag($form);
echo $this->formHidden($form->get('id_artykulu'));
echo $this->formHidden($form->get('data_dodania'));
// echo $this->formHidden($form->get('email_autora'));
echo $this->formRow($form->get('tytul')->setAttribute('class', 'form-control'));
echo $this->formRow($form->get('id_kategorii')->setAttribute('class', 'form-control'));//->setAttribute('options', $data));
echo $this->formRow($form->get('tekst')->setAttribute('class', 'form-control')); ?>

<script>
    CKEDITOR.replace( 'tekst_ckeditor' );
</script>
</br>


<?php echo $this->formSubmit($form->get('submit'));

echo $this->form()->closeTag();
```

Rys. 6 Widok akcji addArticle

8. Baza danych

Ponieważ tworzony moduł ma pobierać artykuły z bazy danych należy utworzyć tabelę `article` o strukturze jak na Rys. 7.

#	Nazwa	Typ	Metoda porównywania napisów	Atrybuty	Null	Ustawienia domyślne	Dodatkowo
1	id_artykulu 	int(10)		UNSIGNED	Nie	Brak	AUTO_INCREMENT
2	data_dodania	datetime			Tak	NULL	
3	tytul	varchar(255)	utf8_bin		Nie	Brak	
4	tekst	longtext	utf8_bin		Tak	NULL	
5	id_kategorii	int(10)		UNSIGNED	Nie	Brak	
6	email_autora	varchar(255)	utf8_bin		Nie	Brak	

Rys. 7 Struktura tabeli z artykułami

9. Model

Zgodnie z wzorcem MVC, za logikę naszego modułu odpowiada model. Aby utworzyć model w aplikacji korzystającej z Zend Framework 2 należy utworzyć w pierwszej kolejności plik `Article.php` w teczce `src/Article/Model`. W pliku tym należy zaimplementować klasę `Article` reprezentującą encję `Article`. Klasa musi posiadać publiczną metodę `exchangeArray()`, służącą do przesyłania danych. Na Rys. 8 przedstawiono implementację omawianej encji dla modułu `Article`.

W kolejnym kroku należy utworzyć, w tej samej teczce, plik `ArticleTable.php`. Powinien on zawierać implementację wzorca projektowego `Table Data Gateway`, umożliwiającego operowanie na danych w bazie. Na Rys. 9 pokazano fragment opisywanej klasy oraz metodę `fetchAll()` realizującą komendę SQL `SELECT ALL`.

```

<?php

namespace Article\Model;

use Zend\InputFilter\InputFilter;
use Zend\InputFilter\InputFilterAwareInterface;
use Zend\InputFilter\InputFilterInterface;

class Article implements InputFilterAwareInterface
{
    public $id_artykulu;
    public $data_dodania;
    public $tytul;
    public $tekst;
    public $id_kategorii;
    public $email_autora;
    protected $inputFilter;

    public function exchangeArray($data)
    {
        $this->id_artykulu = (!empty($data['id_artykulu'])) ? $data['id_artykulu'] : null;
        $this->data_dodania = (!empty($data['data_dodania'])) ? $data['data_dodania'] : null;
        $this->tytul = (!empty($data['tytul'])) ? $data['tytul'] : null;
        $this->tekst = (!empty($data['tekst'])) ? $data['tekst'] : null;
        $this->id_kategorii = (!empty($data['id_kategorii'])) ? $data['id_kategorii'] : null;
        $this->email_autora = (!empty($data['email_autora'])) ? $data['email_autora'] : null;
    }
}

```

Rys. 8 Encja Article

```

<?php

namespace Article\Model;

use Zend\Db\TableGateway\TableGateway;
use Zend\Db\ResultSet\ResultSet;
use Zend\Db\Sql\Select;
use Zend\Paginator\Adapter\DbSelect;
use Zend\Paginator\Paginator;

date_default_timezone_set('Europe/Warsaw');

class ArticleTable
{
    protected $tableGateway;

    public function __construct(TableGateway $tableGateway)
    {
        $this->tableGateway = $tableGateway;
    }

    public function fetchAll()
    {
        $resultSet = $this->tableGateway->select();
        return $resultSet;
    }
}

```

Rys. 9 Fragment ArticleTable.php

10. Formularze

W opisywanym module utworzone zostały formularze służące do obsługi akcji edycji oraz tworzenia nowego artykułu. Aby utworzyć formularz należy stworzyć nowy plik `ArticleForm.php` w teczce `src/Form`. Następnie należy utworzyć klasę dziedziczącą po klasie `Form`. Klasa zawiera tylko jedną metodę o nazwie `__construct`. W jej wnętrzu tworzymy nowe elementy formularza. Dla każdego elementu możemy ustawić odpowiednie parametry tj. `id`, etykieta, widoczność, typ danych, itp.

Na Rys. 10 pokazano fragment utworzonego formularza.

```
<?php

namespace Article\Form;

use Zend\Form\Form;

class ArticleForm extends Form
{
    public function __construct($name = null)
    {
        // we want to ignore the name passed
        parent::__construct('article');

        $this->add(array(
            'name' => 'id_artykulu',
            'type' => 'Hidden',
        ));
        $this->add(array(
            'name' => 'data_dodania',
            'type' => 'Hidden',
        ));
        $this->add(array(
            'name' => 'tytul',
            'type' => 'Text',
            'options' => array(
                'label' => 'Tytuł',
            ),
            'attributes' => array(
                'placeholder' => 'Tytuł artykułu ...',
                'id' => 'tekst_ckeditor2'
            ),
        ));
    }
}
```

Rys. 10 Fragment formularza

11. Walidacja formularzy

Aby zaimplementować walidację danych zapisanych do formularza należy dodać metody `setInputFilter()` oraz `getInputFilter()` do utworzonej w rozdziale wcześniejszym klasie modelu (plik `Article.php` w teczce `src/Article/Model`). Na Rys. 11 zaprezentowano fragment implementacji interfejsu filtrów walidacyjnych.

```
public function setInputFilter(InputFilterInterface $inputFilter)
{
    throw new \Exception("Not used");
}

public function getInputFilter()
{
    if (!$this->inputFilter) {
        $inputFilter = new InputFilter();

        $inputFilter->add(array(
            'name'      => 'id_artykułu',
            'required'  => true,
            'filters'   => array(
                array('name' => 'Int'),
            ),
        ));

        $inputFilter->add(array(
            'name'      => 'tekst',
            'required'  => true,
            'filters'   => array(
                //array('name' => 'StripTags'),
                array('name' => 'StringTrim'),
            ),
            'validators' => array(
                array(
                    'name'      => 'StringLength',
                    'options' => array(
                        'encoding' => 'UTF-8',
                        'min'      => 1,
                        'max'      => 10000000,
                    ),
                ),
            ),
        ));
    }
}
```

Rys. 11 Fragment walidacji

12. Autoryzacja

Aby zrealizować podział na trzech różnych aktorów, posiadających odmienne uprawnienia należy zaimplementować autoryzację. W tym celu wykorzystano gotowy moduł BjjAuthorize (należy go pobrać poprzez composera, a następnie włączyć).

Moduł ten pozwala na pobranie z bazy uprawnień aktora, a następnie zablokowanie tych akcji, które wymagają wyższych uprawnień. Ustawienia strażnika dla opisywanej aplikacji zaprezentowano na Rys. 12.

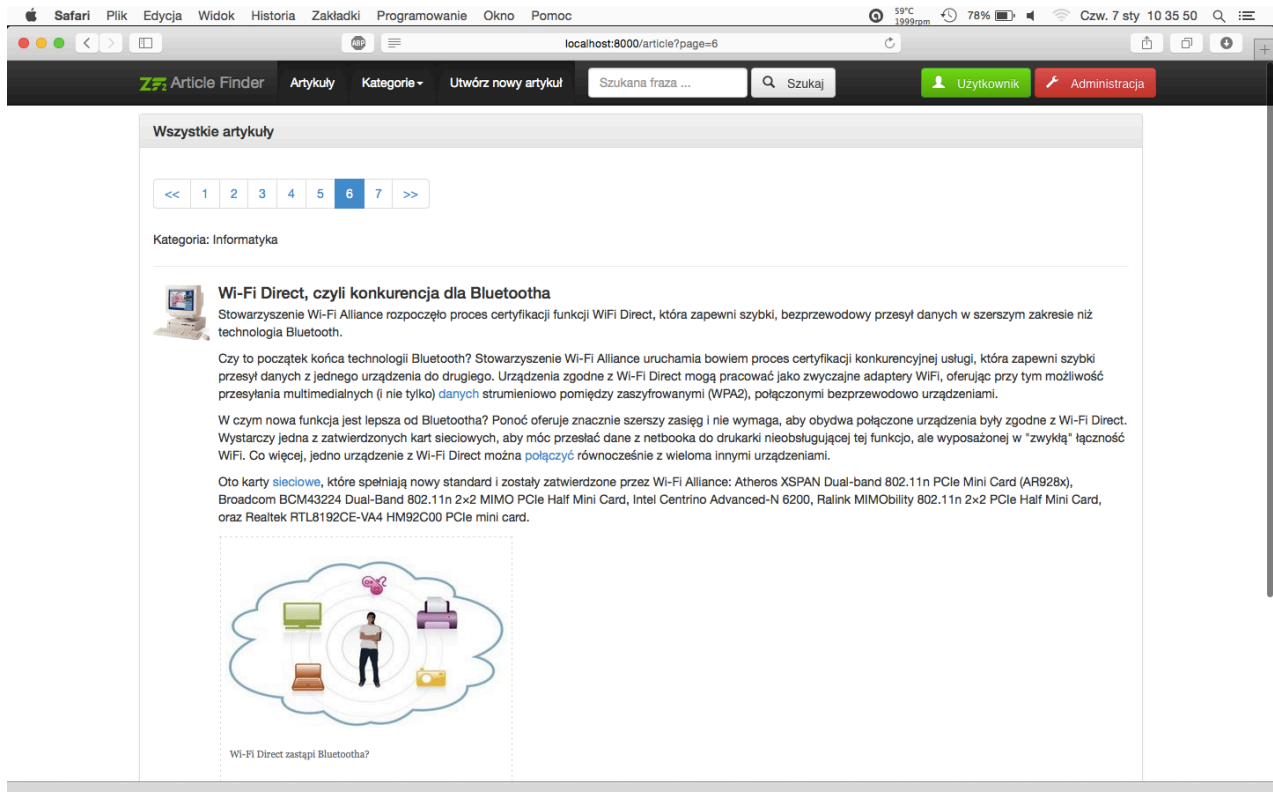
```
'guards' => array(
    /* If this guard is specified here (i.e. it is enabled), it will block
     * access to all controllers and actions unless they are specified here.
     * You may omit the 'action' index to allow access to the entire controller
     */
    'BjjAuthorize\Guard\Controller' => array(
        array('controller' => 'index', 'action' => 'index', 'roles' => array('guest', 'user')),
        array('controller' => 'Article\Controller\Article', 'action' => 'index', 'roles' => array('guest', 'user')),

        array('controller' => 'Article\Controller\Article', 'action' => 'biology', 'roles' => array('guest', 'user')),
        array('controller' => 'Article\Controller\Article', 'action' => 'electronics', 'roles' => array('guest', 'user')),
        array('controller' => 'Article\Controller\Article', 'action' => 'informatics', 'roles' => array('guest', 'user')),
        array('controller' => 'Article\Controller\Article', 'action' => 'math', 'roles' => array('guest', 'user')),
        array('controller' => 'Article\Controller\Article', 'action' => 'physics', 'roles' => array('guest', 'user')),
        array('controller' => 'Article\Controller\Article', 'action' => 'other', 'roles' => array('guest', 'user')),

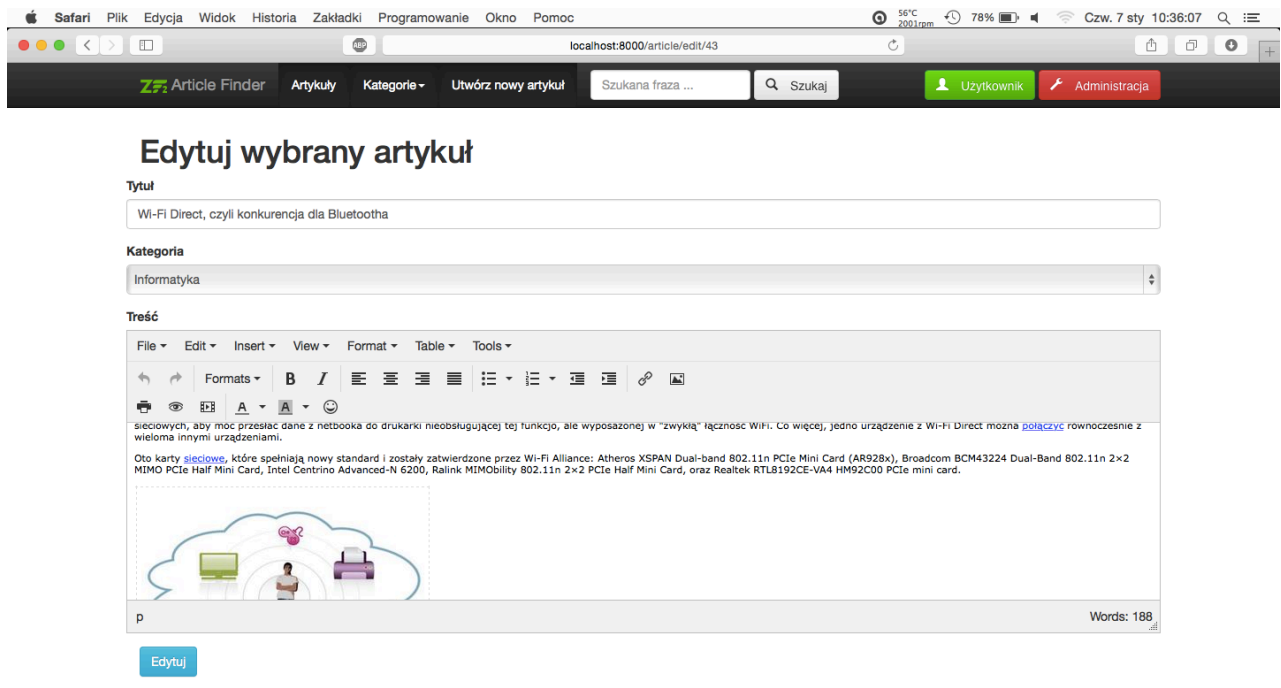
        array('controller' => 'Article\Controller\Article', 'action' => 'add', 'roles' => array('user')),
        array('controller' => 'Article\Controller\Article', 'action' => 'edit', 'roles' => array('user')),
        array('controller' => 'Article\Controller\Article', 'action' => 'delete', 'roles' => array('user')),
        array('controller' => 'Article\Controller\Article', 'action' => 'admin', 'roles' => array('admin')),
    )
)
```

Rys. 12 Ustawienia strażnika dla modułu Article

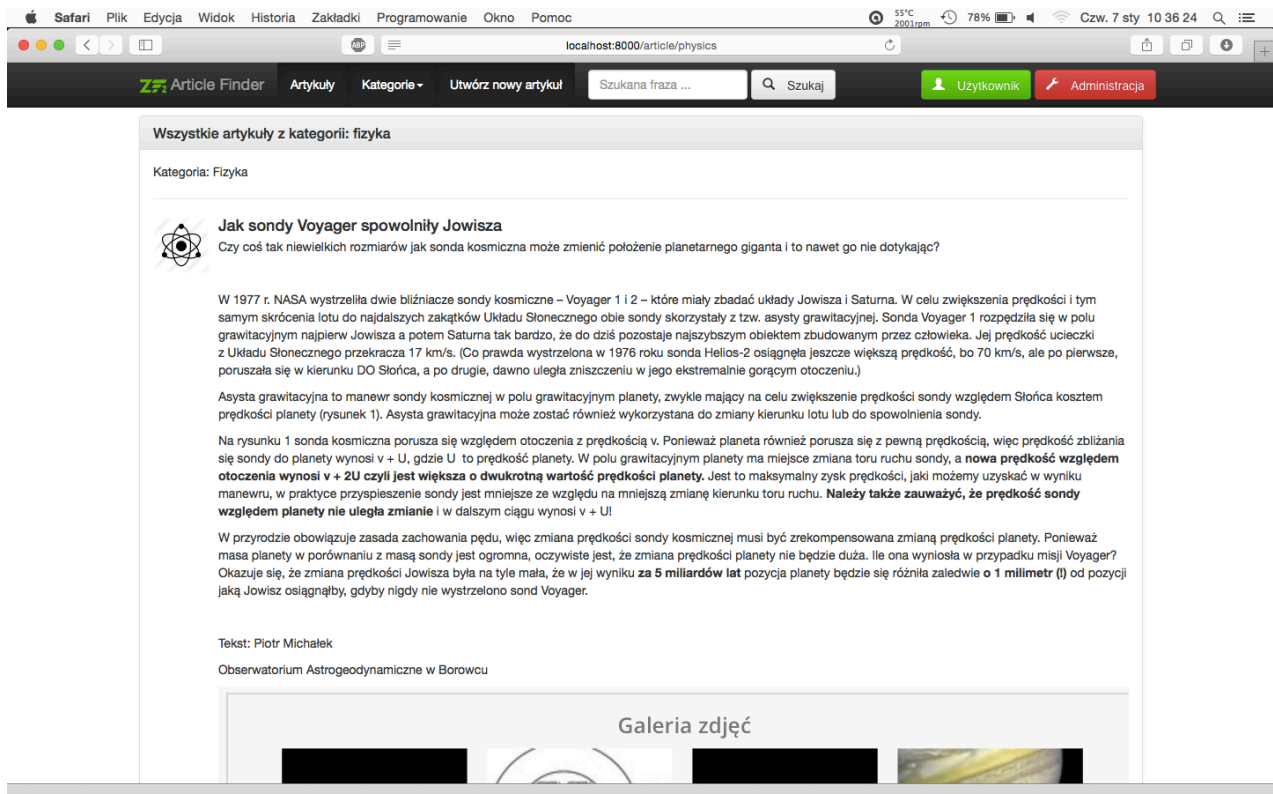
Prezentacja działania modułu



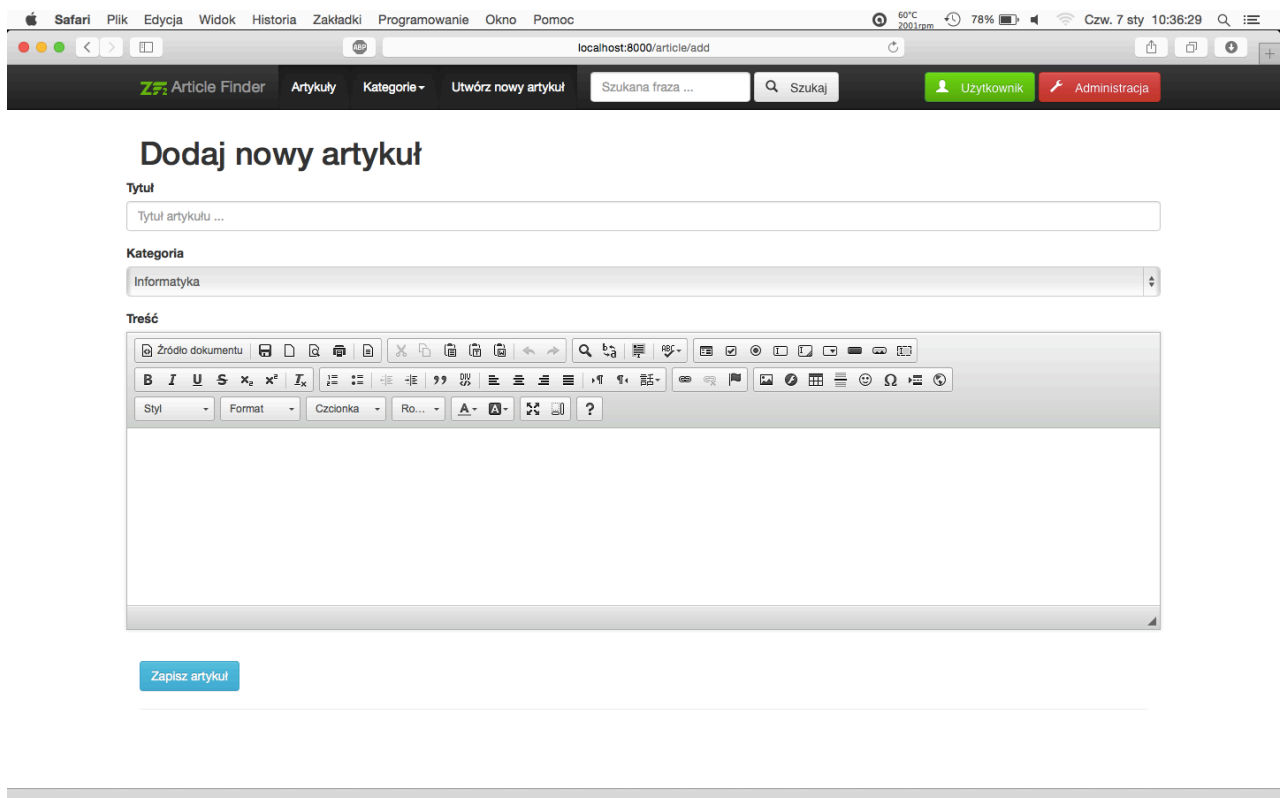
Rys. 13 indexAction



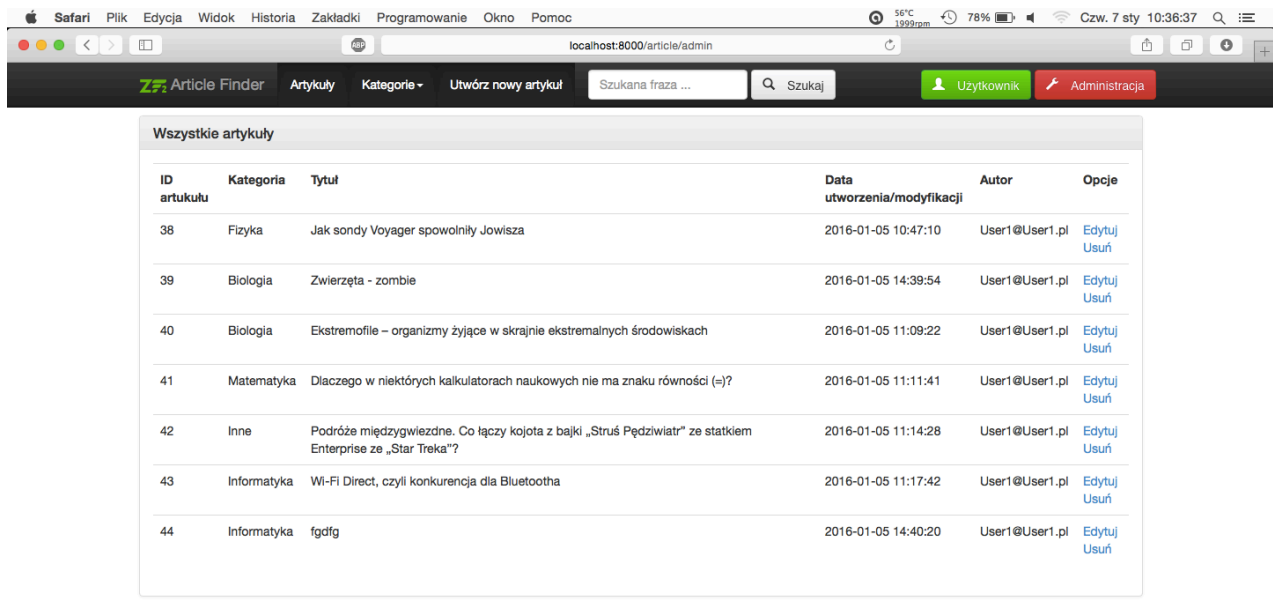
Rys. 14 editAction



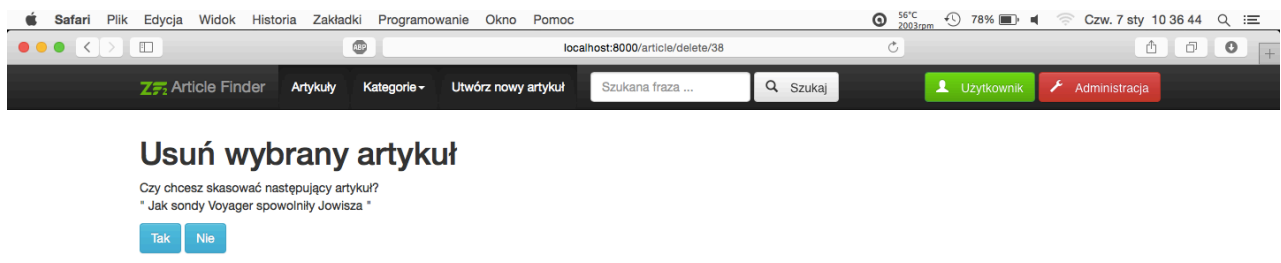
Rys. 15 physicsAction



Rys. 16 addAction



Rys. 17 adminAction



Rys. 18 deleteAction