Contents

Introduction to Laminas Framework

Installation and Setup

Laminas requires PHP ≥ 8.1, Composer, and web server support (Apache, Nginx, or PHP built-in server).

Check PHP Version

```
php -v
```

Check Composer Installation

```
composer --version
```

Start PHP Built-in Server (after project setup)

```
php -S localhost:8080 -t public
```

Installing Laminas using Composer

Create a new project with MVC skeleton

```
composer create-project laminas/laminas-mvc-skeleton myapp
```

Install a specific component (e.g., Laminas Router)

```
composer require laminas/laminas-router
```

Update dependencies

```
composer update
```

Setting up a Laminas project

Once installed, the structure includes:

```
module/ → Your application logic
public/ → Web root
config/ → Application config
```

Run your app

```
php -S 0.0.0.0:8080 -t public
```

Create a new module (e.g., Blog)

```
php vendor/bin/laminas module:create Blog
```

Enable module in config/modules.config.php

```
return [
   'Application',
   'Blog', // newly added
];
```

MVC Architecture in Laminas

Understanding MVC in Laminas

MVC = Model–View–Controller, which Laminas strictly enforces.

- Model → Business logic/data
- View → Templates (HTML/PHTML)
- Controller → Handles requests and passes data

Example Flow
User visits /blog → Router → BlogController → BlogModel → blog/index.phtml

Diagram (conceptual)
Request → Router → Controller → Model → View → Response

Controllers, views, and models

1. Controller Example

module/Blog/src/Controller/BlogController.php

```
namespace Blog\Controller;

use Laminas\Mvc\Controller\AbstractActionController;
use Laminas\View\Model\ViewModel;

class BlogController extends AbstractActionController {
   public function indexAction() {
      return new ViewModel(['posts' => ['First Post', 'Second Post']]);
   }
}
```

2. View Example

module/Blog/view/blog/blog/index.phtml

```
<h2>Blog Posts</h2>
<ul>
<?php foreach ($this->posts as $post): ?>
   <li><?= $this->escapeHtml($post) ?></li>
<?php endforeach; ?>
</ul>
```

3. Model Example

module/Blog/src/Model/Post.php

```
namespace Blog\Model;

class Post {
   public $id;
   public $title;
```

```
    public function __construct($id, $title) {
        $this->id = $id;
        $this->title = $title;
    }
}
```

Defining routes in Laminas

Routes tell Laminas how to map URLs to controllers.

Simple Literal Route

```
'router' => [
    'routes' => [
        'home' => [
            'type' => 'Literal',
            'options' => [
                'route'    => '/',
                'defaults' => [
                    'controller' => Controller\IndexController::class,
                    'action'     => 'index',
                ],
            ],
        ],
    ],
],
```

Segment Route (with parameters)

```
'blog' => [
    'type' => 'Segment',
    'options' => [
        'route' => '/blog[/:id]',
        'defaults' => [
            'controller' => Controller\BlogController::class,
            'action' => 'view',
        ],
    ],
],
```

Wildcard Route

```
'admin' => [
    'type' => 'Segment',
    'options' => [
        'route' => '/admin[/:controller[/:action]]',
        'defaults' => [
            'controller' => Controller\AdminController::class,
            'action' => 'index',
        ],
    ],
],
```

Handling HTTP requests and responses

Laminas provides Laminas\Http\Request and Laminas\Http\Response.

Getting Query Parameters

```
$id = $this->params()->fromQuery('id', 1);
```

Getting POST Data

```
$name = $this->params()->fromPost('name', 'Guest');
```

Returning a JSON Response

```
use Laminas\View\Model\JsonModel;

public function apiAction() {
   return new JsonModel(['status' => 'success', 'data' => [1,2,3]]);
}
```

Custom Response Example

```
$response = $this->getResponse();
$response->setContent("Custom Response Body");
return $response;
```

Project 1: Mini-Blog App

1. Installation and Project Setup

First, we need to install Laminas MVC skeleton.

```
composer create-project laminas/laminas-mvc-skeleton blog-app
cd blog-app
php -S 0.0.0.0:8080 -t public
```

Visit http://localhost:8080 and you should see the default Laminas welcome page.

2. Create a Blog Module

We'll keep our blog code inside a new module named Blog.

```
php vendor/bin/laminas module:create Blog
```

This creates:

- module/Blog/config/
- module/Blog/src/
- module/Blog/view/

Enable the module (Edit config/modules.config.php):

```
return [
   'Laminas\Router',
   'Laminas\Validator',
   'Application',
   'Blog', // add here
];
```

3. Define Routes for Blog

Open module/Blog/config/module.config.php and add routes:

```
'router' => [
    'routes' => [
        'blog' => [
            'type' => 'Segment',
            'options' => [
                'route' => '/blog[/:action[/:id]]',
                'defaults' => [
                    'controller' => Blog\Controller\BlogController::class,
                    'action' => 'index',
                ],
            ],
        ],
    ],
],
```

Now, /blog, /blog/view/1, /blog/add will all map to the BlogController.

4. Create a Blog Controller

Make a new controller class.
module/Blog/src/Controller/BlogController.php

```
namespace Blog\Controller;

use Laminas\Mvc\Controller\AbstractActionController;
use Laminas\View\Model\ViewModel;
use Blog\Model\Post;

class BlogController extends AbstractActionController
{
    // Display list of posts
    public function indexAction()
    {
        $posts = [
            new Post(1, "Hello World", "This is my first blog post."),
            new Post(2, "Second Post", "Laminas Framework makes PHP fun!"),
        ];
        return new ViewModel(['posts' => $posts]);
    }

    // View a single post
    public function viewAction()
    {
        $id = (int) $this->params()->fromRoute('id', 0);
        $post = new Post($id, "Sample Post #$id", "Content for post #$id");
        return new ViewModel(['post' => $post]);
    }
```

```
    // Add a new post (dummy for now)
    public function addAction()
    {
        return new ViewModel(['message' => "Form to add a new post"]);
    }
}
```

5. Create a Post Model

module/Blog/src/Model/Post.php

```
namespace Blog\Model;

class Post
{
    public $id;
    public $title;
    public $content;

    public function __construct($id, $title, $content)
    {
        $this->id      = $id;
        $this->title   = $title;
        $this->content = $content;
    }
}
```

6. Create Views

Now we'll add views for each action.

module/Blog/view/blog/blog/index.phtml

```
<h2>Blog Posts</h2>
<ul>
<?php foreach ($this->posts as $post): ?>
    <li>
        <a href="<?= $this->url('blog', ['action' => 'view', 'id' => $post->id]) ?>">
            <?= $this->escapeHtml($post->title) ?>
        </a>
    </li>
<?php endforeach; ?>
</ul>
<a href="<?= $this->url('blog', ['action' => 'add']) ?>">Add New Post</a>
```

module/Blog/view/blog/blog/view.phtml

```
<h2><?= $this->escapeHtml($this->post->title) ?></h2>
<p><?= $this->escapeHtml($this->post->content) ?></p>
<a href="<?= $this->url('blog') ?>">Back to Posts</a>
```

module/Blog/view/blog/blog/add.phtml

```
<h2>Add a New Post</h2>
<p><?= $this->message ?></p>
<form method="post">
  <label>Title: <input type="text" name="title"></label><br>
  <label>Content: <textarea name="content"></textarea></label><br>
  <button type="submit">Save</button>
</form>
```

7. Handling HTTP Requests & Responses

We can now handle form submissions inside addAction.

Update BlogController.php

```
public function addAction()
{
  $request = $this->getRequest();

  if ($request->isPost()) {
    $title = $this->params()->fromPost('title', 'Untitled');
    $content = $this->params()->fromPost('content', 'No content');

    // Normally, we'd save to DB. For now, return a confirmation.
    return new ViewModel([
      'message' => "New post created: $title",
    ]);
  }

  return new ViewModel();
}
```

8. Testing Our Blog

1. Visit http://localhost:8080/blog → list of posts.
2. Click a post → http://localhost:8080/blog/view/1 shows details.
3. Add a new post → http://localhost:8080/blog/add shows form.

9. OPTIONAL 1: Connecting and saving the blog to a database

Step 1: Install Laminas\Db

```
composer require laminas/laminas-db
```

Step 2: Create Database & Table

MySQL Example

```
CREATE DATABASE laminas_blog;
USE laminas_blog;

CREATE TABLE posts (
  id INT AUTO_INCREMENT PRIMARY KEY,
```

```
  title VARCHAR(255) NOT NULL,
  content TEXT NOT NULL
);
```

PostgreSQL Example

```
CREATE DATABASE laminas_blog;

\c laminas_blog;

CREATE TABLE posts (
  id SERIAL PRIMARY KEY,
  title VARCHAR(255) NOT NULL,
  content TEXT NOT NULL
);
```

Step 3: Configure Database Adapter

Edit config/autoload/global.php (create if not exists):

```
return [
  'db' => [
    'driver'   => 'Pdo',
    'dsn'      => 'mysql:dbname=laminas_blog;host=localhost;charset=utf8',
    // For PostgreSQL use: 'pgsql:host=localhost;port=5432;dbname=laminas_blog'
    'username' => 'root',   // or your DB user
    'password' => 'secret', // your DB password
  ],
  'service_manager' => [
    'factories' => [
      Laminas\Db\Adapter\Adapter::class => Laminas\Db\Adapter\AdapterServiceFactory::class,
    ],
  ],
];
```

Step 4: Create a PostTable Class

module/Blog/src/Model/PostTable.php

```
namespace Blog\Model;

use Laminas\Db\TableGateway\TableGatewayInterface;

class PostTable
{
  private $tableGateway;

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

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

    public function getPost($id)
    {
        $id = (int) $id;
        $rowset = $this->tableGateway->select(['id' => $id]);
        $row = $rowset->current();
        if (!$row) {
            throw new \Exception("Could not find row $id");
        }
        return $row;
    }

    public function savePost(Post $post)
    {
        $data = [
            'title'   => $post->title,
            'content' => $post->content,
        ];

        if ($post->id) {
            $this->tableGateway->update($data, ['id' => $post->id]);
        } else {
            $this->tableGateway->insert($data);
            $post->id = $this->tableGateway->getLastInsertValue();
        }
    }
}
```

Step 5: Update Post Model

module/Blog/src/Model/Post.php

```
namespace Blog\Model;

class Post
{
    public $id;
    public $title;
    public $content;

    public function exchangeArray(array $data)
    {
        $this->id     = !empty($data['id']) ? (int) $data['id'] : null;
        $this->title  = $data['title'] ?? null;
        $this->content = $data['content'] ?? null;
    }
}
```

Step 6: Wire Up Dependencies

Modify module/Blog/config/module.config.php to tell Laminas how to create services.

```
use Blog\Model\Post;
use Blog\Model\PostTable;
use Laminas\Db\TableGateway\TableGateway;
use Laminas\Db\ResultSet\ResultSet;

return [
  'service_manager' => [
    'factories' => [
      PostTable::class => function($container) {
        $dbAdapter = $container->get(\Laminas\Db\Adapter\Adapter::class);
        $resultSetPrototype = new ResultSet();
        $resultSetPrototype->setArrayObjectPrototype(new Post());
        $tableGateway = new TableGateway('posts', $dbAdapter, null, $resultSetPrototype);
        return new PostTable($tableGateway);
      },
    ],
  ],
];
```

Step 7: Update BlogController

module/Blog/src/Controller/BlogController.php

```
namespace Blog\Controller;

use Laminas\Mvc\Controller\AbstractActionController;
use Laminas\View\Model\ViewModel;
use Blog\Model\Post;
use Blog\Model\PostTable;

class BlogController extends AbstractActionController
{
  private $postTable;

  public function __construct(PostTable $postTable)
  {
    $this->postTable = $postTable;
  }

  public function indexAction()
  {
    $posts = $this->postTable->fetchAll();
    return new ViewModel(['posts' => $posts]);
  }

  public function viewAction()
  {
    $id = (int) $this->params()->fromRoute('id', 0);
```

```
        try {
            $post = $this->postTable->getPost($id);
        } catch (\Exception $e) {
            return $this->redirect()->toRoute('blog');
        }
        return new ViewModel(['post' => $post]);
    }

    public function addAction()
    {
        $request = $this->getRequest();

        if ($request->isPost()) {
            $title   = $this->params()->fromPost('title');
            $content = $this->params()->fromPost('content');

            $post = new Post();
            $post->exchangeArray(['title' => $title, 'content' => $content]);
            $this->postTable->savePost($post);

            return $this->redirect()->toRoute('blog');
        }

        return new ViewModel();
    }
}
```

Configure the controller factory so Laminas knows how to inject PostTable.

module/Blog/config/module.config.php → add:

```
'controllers' => [
    'factories' => [
        Blog\Controller\BlogController::class => function($container) {
            return new Blog\Controller\BlogController(
                $container->get(Blog\Model\PostTable::class)
            );
        },
    ],
],
```

Step 8: Update Views

We can reuse our existing views:

index.phtml

```
<h2>Blog Posts</h2>
<ul>
<?php foreach ($this->posts as $post): ?>
    <li>
        <a href="<?= $this->url('blog', ['action' => 'view', 'id' => $post->id]) ?>">
```

```
        <?= $this->escapeHtml($post->title) ?>
      </a>
    </li>
<?php endforeach; ?>
</ul>
<a href="<?= $this->url('blog', ['action' => 'add']) ?>">Add New Post</a>
```

view.phtml

```
<h2><?= $this->escapeHtml($this->post->title) ?></h2>
<p><?= $this->escapeHtml($this->post->content) ?></p>
<a href="<?= $this->url('blog') ?>">Back to Posts</a>
```

add.phtml

```
<h2>Add a New Post</h2>
<form method="post">
  <label>Title: <input type="text" name="title"></label><br>
  <label>Content: <textarea name="content"></textarea></label><br>
  <button type="submit">Save</button>
</form>
```

10. OPTIONAL 2: Mini-Blog Full CRUD

Step 1: Update PostTable with Update & Delete Methods

module/Blog/src/Model/PostTable.php

```
public function deletePost($id)
{
  $this->tableGateway->delete(['id' => (int) $id]);
}
```

(The savePost() we already wrote handles both insert and update.)

Step 2: Add Edit and Delete Actions in BlogController

module/Blog/src/Controller/BlogController.php

```
public function editAction()
{
  $id = (int) $this->params()->fromRoute('id', 0);

  if ($id === 0) {
    return $this->redirect()->toRoute('blog', ['action' => 'add']);
  }

  try {
    $post = $this->postTable->getPost($id);
  } catch (\Exception $e) {
    return $this->redirect()->toRoute('blog');
  }
```

```
    $request = $this->getRequest();

    if ($request->isPost()) {
        $title   = $this->params()->fromPost('title');
        $content = $this->params()->fromPost('content');

        $post->exchangeArray([
            'id'     => $id,
            'title'  => $title,
            'content' => $content,
        ]);

        $this->postTable->savePost($post);

        return $this->redirect()->toRoute('blog');
    }

    return new ViewModel(['post' => $post]);
}

public function deleteAction()
{
    $id = (int) $this->params()->fromRoute('id', 0);

    if ($id === 0) {
        return $this->redirect()->toRoute('blog');
    }

    $request = $this->getRequest();

    if ($request->isPost()) {
        $del = $this->params()->fromPost('del', 'No');

        if ($del === 'Yes') {
            $id = (int) $this->params()->fromPost('id');
            $this->postTable->deletePost($id);
        }

        return $this->redirect()->toRoute('blog');
    }

    return new ViewModel([
        'id'   => $id,
        'post' => $this->postTable->getPost($id),
    ]);
}
```

Step 3: Update Routes to Allow Edit/Delete

module/Blog/config/module.config.php

```
'router' => [
  'routes' => [
    'blog' => [
      'type' => 'Segment',
      'options' => [
        'route' => '/blog[/:action[/:id]]',
        'defaults' => [
          'controller' => Blog\Controller\BlogController::class,
          'action'    => 'index',
        ],
      ],
    ],
  ],
],
```

(We already had this, but now :action will also match edit and delete.)

Step 4: Update Views

index.phtml (Add Edit/Delete links)

```
<h2>Blog Posts</h2>
<ul>
<?php foreach ($this->posts as $post): ?>
  <li>
    <a href="<?= $this->url('blog', ['action' => 'view', 'id' => $post->id]) ?>">
      <?= $this->escapeHtml($post->title) ?>
    </a>
    |
    <a href="<?= $this->url('blog', ['action' => 'edit', 'id' => $post->id]) ?>">Edit</a>
    |
    <a href="<?= $this->url('blog', ['action' => 'delete', 'id' => $post->id]) ?>">Delete</a>
  </li>
<?php endforeach; ?>
</ul>
<a href="<?= $this->url('blog', ['action' => 'add']) ?>">Add New Post</a>
```

edit.phtml

```
<h2>Edit Post</h2>
<form method="post">
  <input type="hidden" name="id" value="<?= $this->post->id ?>">
  <label>Title: <input type="text" name="title" value="<?= $this->escapeHtml($this->post->title) ?>"></label><br>
  <label>Content: <textarea name="content"><?= $this->escapeHtml($this->post->content)
?></textarea></label><br>
  <button type="submit">Save Changes</button>
</form>
<a href="<?= $this->url('blog') ?>">Cancel</a>
```

delete.phtml

```
<h2>Delete Post</h2>
<p>Are you sure you want to delete "<strong><?= $this->escapeHtml($this->post->title) ?></strong>"?</p>
<form method="post">
  <input type="hidden" name="id" value="<?= $this->id ?>">
  <button type="submit" name="del" value="Yes">Yes, delete</button>
  <button type="submit" name="del" value="No">No, cancel</button>
</form>
```

Endpoints:

Create	/blog/add	(adds new post)
Read	/blog, /blog/view/:id	(list & view posts)
Update	/blog/edit/:id	(edit existing post)
Delete	/blog/delete/:id	(delete confirmation)

Creating Forms with Laminas\Form\Form

Using Form class to create form structures

In Laminas, the Laminas\Form\Form class lets you define reusable, object-oriented form structures. Instead of writing HTML manually, you build forms programmatically.

Creating a Simple Form

```
use Laminas\Form\Form;

$form = new Form('user-form');
```

Add a field later

```
$form->add([
  'name' => 'username',
  'type' => 'Text',
]);
```

Render form in a view

```
echo $this->form()->openTag($form);
echo $this->formRow($form->get('username'));
echo $this->form()->closeTag();
```

Assign method and action

```
$form->setAttribute('method', 'post');
$form->setAttribute('action', '/register');
```

Adding form elements (Text, Email, Password, Submit)

Laminas provides element types like Text, Email, Password, and Submit.

Text Field

```
$form->add([
   'name' => 'username',
   'type' => 'Text',
   'options' => ['label' => 'Username'],
]);
```

Email Field

```
$form->add([
   'name' => 'email',
   'type' => 'Email',
   'options' => ['label' => 'Email Address'],
]);
```

Password Field

```
$form->add([
   'name' => 'password',
   'type' => 'Password',
   'options' => ['label' => 'Password'],
]);
```

Submit Button

```
$form->add([
   'name' => 'submit',
   'type' => 'Submit',
   'attributes' => ['value' => 'Register'],
]);
```

Setting element attributes and options

Attributes affect HTML rendering, while options control Laminas-specific behavior (labels, validators, etc.).

Placeholder

```
$form->get('username')->setAttribute('placeholder', 'Enter username');
```

CSS Class

```
$form->get('email')->setAttribute('class', 'form-control');
```

Set ID

```
$form->get('password')->setAttribute('id', 'pwd-field');
```

Label Option

```
$form->get('username')->setOptions(['label' => 'Your Username']);
```

Introduction to InputFilter and InputFactory

An InputFilter defines validation and filtering rules for form inputs. The InputFactory helps create them dynamically.

Create InputFilter

```
use Laminas\InputFilter\InputFilter;

$inputFilter = new InputFilter();
```

Attach to Form

```
$form->setInputFilter($inputFilter);
```

Using InputFactory

```
use Laminas\InputFilter\Factory as InputFactory;

$factory = new InputFactory();
$input = $factory->createInput([
  'name' => 'username',
  'required' => true,
]);
$inputFilter->add($input);
```

Validation Workflow

```
$form->setData($_POST);
if ($form->isValid()) {
  $data = $form->getData();
}
```

Adding validators (e.g., StringLength, EmailAddress)

Validators ensure inputs meet rules.

String Length

```
$inputFilter->add([
  'name' => 'username',
  'validators' => [
    ['name' => 'StringLength', 'options' => ['min' => 3, 'max' => 20]],
  ],
]);
```

Email Address

```
$inputFilter->add([
  'name' => 'email',
  'validators' => [
    ['name' => 'EmailAddress'],
  ],
]);
```

Regex Validator

```
$inputFilter->add([
   'name' => 'password',
   'validators' => [
      ['name' => 'Regex', 'options' => ['pattern' => '/^(?=.*[A-Z]).+$/']],
   ],
]);
```

Filtering input (e.g., StringTrim, StripTags)

Filters modify user input before validation (e.g., trimming whitespace, stripping HTML).

Trim Whitespace

```
'filters' => [['name' => 'StringTrim']],
```

Remove HTML Tags

```
'filters' => [['name' => 'StripTags']],
```

Convert to Lowercase

```
'filters' => [['name' => 'StringToLower']],
```

Example with Validator

```
$inputFilter->add([
   'name' => 'username',
   'filters' => [
      ['name' => 'StringTrim'],
      ['name' => 'StripTags'],
   ],
   'validators' => [
      ['name' => 'StringLength', 'options' => ['min' => 3]],
   ],
]);
```

Using InputFilterProviderInterface

Instead of defining InputFilter separately, you can make your form implement InputFilterProviderInterface so validation rules live in the form class.

Example Form Class

```
use Laminas\Form\Form;
use Laminas\InputFilter\InputFilterProviderInterface;

class RegisterForm extends Form implements InputFilterProviderInterface
{
   public function __construct()
   {
      parent::__construct('register');
```

```
        $this->add(['name' => 'username', 'type' => 'Text']);
        $this->add(['name' => 'email', 'type' => 'Email']);
        $this->add(['name' => 'password', 'type' => 'Password']);
        $this->add(['name' => 'submit', 'type' => 'Submit']);
    }

    public function getInputFilterSpecification()
    {
        return [
            'username' => [
                'required' => true,
                'filters'  => [['name' => 'StringTrim'], ['name' => 'StripTags']],
                'validators' => [['name' => 'StringLength', 'options' => ['min' => 3, 'max' => 20]]],
            ],
            'email' => [
                'required' => true,
                'validators' => [['name' => 'EmailAddress']],
            ],
        ];
    }
}
```

Concept of entity binding in Laminas

Entity binding links a form to an object (entity). This allows form fields to populate object properties automatically and vice versa.

Bind a Post object to a form

```
$post = new \Blog\Model\Post();
$form->bind($post);
```

Form auto-populates from entity

```
$post->title = "Hello Laminas!";
$form->bind($post);
echo $form->get('title')->getValue(); // Hello Laminas!
```

Entity updated on form submit

```
$form->setData($_POST);
if ($form->isValid()) {
    // $post object is automatically updated
    echo $post->title;
}
```

Using hydrators like ClassMethods, ObjectProperty

A hydrator transfers data between arrays and objects.

- ClassMethods → uses getter/setter methods.
- ObjectProperty → accesses public properties directly.

Using ClassMethods Hydrator

```
use Laminas\Hydrator\ClassMethodsHydrator;

$form->setHydrator(new ClassMethodsHydrator());
$form->bind(new Post());
```

Using ObjectProperty Hydrator

```
use Laminas\Hydrator\ObjectPropertyHydrator;

$form->setHydrator(new ObjectPropertyHydrator());
$form->bind(new Post());
```

Extracting data from entity

```
$hydrator = new ClassMethodsHydrator();
$data = $hydrator->extract($post); // ['id' => 1, 'title' => 'Hello']
```

Hydrating entity from array

```
$hydrator->hydrate(['title' => 'New Title'], $post);
```

Creating reusable form models

Instead of defining forms in controllers, you build dedicated form classes that can be reused across controllers/views.

Define a PostForm class

```
namespace Blog\Form;

use Laminas\Form\Form;

class PostForm extends Form {
   public function __construct() {
      parent::__construct('post-form');
      $this->add(['name' => 'title', 'type' => 'Text']);
      $this->add(['name' => 'content', 'type' => 'Textarea']);
      $this->add(['name' => 'submit', 'type' => 'Submit']);
   }
}
```

Use in Controller

```
$form = $this->getForm(PostForm::class);
```

Re-use for Add/Edit

```
// Add action
$form = new PostForm();
$form->bind(new Post());

// Edit action
$form->bind($existingPost);
```

Populating forms with object data

Binding automatically populates form fields with entity data.

Populate from entity

```
$post->title = "Update Me!";
$form->bind($post);
```

Set default values

```
$form->get('title')->setValue('Default Title');
```

Populate from array

```
$form->setData(['title' => 'Array Title', 'content' => 'Sample']);
```

Re-populate after validation error

```
$form->setData($_POST);
$form->isValid(); // keeps entered values visible
```

Setting up controller actions for form handling

Controller actions handle form rendering, submission, and persistence.

Example: BlogController

```
public function addAction() {
   $form = new PostForm();
   $form->bind(new Post());

   $request = $this->getRequest();
   if ($request->isPost()) {
      $form->setData($request->getPost());
      if ($form->isValid()) {
         $this->postTable->savePost($form->getData());
         return $this->redirect()->toRoute('blog');
      }
   }
   return ['form' => $form];
}
```

Validating POST requests and displaying errors

Validation rules (via InputFilter) prevent invalid data from saving.

Check if valid

```
if (!$form->isValid()) {
   $messages = $form->getMessages();
}
```

Display error in view

```
echo $this->formElementErrors($form->get('title'));
```

Force required field

```
$form->getInputFilter()->get('title')->setRequired(true);
```

Loop through errors

```
foreach ($messages as $field => $error) {
   echo "$field: " . implode(', ', $error);
}
```

Redirecting after success or failure

After submission, you typically redirect to avoid duplicate form submissions.

Redirect on success

```
return $this->redirect()->toRoute('blog');
```

Redirect with params

```
return $this->redirect()->toRoute('blog', ['action' => 'view', 'id' => $id]);
```

Stay on same page (failure)

```
return new ViewModel(['form' => $form]);
```

Customizing action responses

You can return different response types, not just HTML.

JSON response

```
use Laminas\View\Model\JsonModel;

return new JsonModel(['status' => 'success']);
```

Custom HTTP response

```
$response = $this->getResponse();
$response->setStatusCode(403);
$response->setContent('Forbidden');
return $response;
```

Redirect with flash message

```
$this->flashMessenger()->addSuccessMessage("Post added successfully!");
return $this->redirect()->toRoute('blog');
```

Partial ViewModel

```
return (new ViewModel(['form' => $form]))->setTerminal(true);
```

Creating forms using Fieldset and Collection

- Fieldset: A reusable sub-form (like grouping related inputs).
- Collection: Allows repeating groups of inputs (e.g., multiple addresses).

Define a Fieldset for an Address

```
use Laminas\Form\Fieldset;
use Laminas\Form\Element;

class AddressFieldset extends Fieldset {
  public function __construct() {
    parent::__construct('address');

    $this->add(['name' => 'street', 'type' => Element\Text::class, 'options' => ['label' => 'Street']]);
    $this->add(['name' => 'city', 'type' => Element\Text::class, 'options' => ['label' => 'City']]);
  }
}
```

Use Fieldset in a Form

```
$form->add([
  'type' => AddressFieldset::class,
  'name' => 'home_address',
]);
```

Use Collection for Multiple Addresses

```
use Laminas\Form\Element\Collection;

$form->add([
  'type' => Collection::class,
  'name' => 'addresses',
  'options' => [
    'count' => 2,
    'target_element' => new AddressFieldset(),
  ],
]);
```

Bind with Entity

```
$form->bind(new UserEntity()); // User has addresses[]
```

Enabling CSRF elements in forms

CSRF (Cross-Site Request Forgery) protection prevents malicious form submissions by requiring a hidden token.

Add CSRF to Form

```
$form->add([
  'type' => 'Csrf',
  'name' => 'csrf',
]);
```

Custom CSRF Options

```
$form->add([
   'type' => 'Csrf',
   'name' => 'csrf',
   'options' => ['csrf_options' => ['timeout' => 600]],
]);
```

Render in View

```
echo $this->formRow($form->get('csrf'));
```

Validate Automatically

- When $form->isValid() is called, CSRF is checked. If invalid → error.

Using Laminas\Form\Element\Csrf

This is the specific CSRF element class.

Basic Usage

```
use Laminas\Form\Element\Csrf;

$csrf = new Csrf('security');
$form->add($csrf);
```

Custom Error Message

```
$form->get('security')->setOptions([
   'csrf_options' => [
      'messages' => [
         Laminas\Validator\Csrf::NOT_SAME => 'Invalid security token',
      ],
   ],
]);
```

Set Timeout

```
$form->get('security')->setOptions([
   'csrf_options' => ['timeout' => 300],
]);
```

Database Interaction

Connecting to a database in Laminas

Use Laminas\Db\Adapter\Adapter for database connections.

Configuration in global.php

```
'db' => [
   'driver' => 'Pdo_Mysql',
   'dsn' => 'mysql:dbname=laminas_blog;host=localhost;charset=utf8',
   'username' => 'root',
   'password' => 'secret',
],
'service_manager' => [
   'factories' => [
      Laminas\Db\Adapter\Adapter::class => Laminas\Db\Adapter\AdapterServiceFactory::class,
   ],
],
```

Get Adapter in Controller

```
$adapter = $this->getEvent()->getApplication()->getServiceManager()->get(Adapter::class);
```

Run Query

```
$statement = $adapter->query("SELECT * FROM posts");
$result = $statement->execute();
```

Prepared Statement

```
$statement = $adapter->createStatement("SELECT * FROM posts WHERE id = ?", [1]);
$result = $statement->execute();
```

Performing CRUD operations

Using TableGateway simplifies CRUD in Laminas.

Setup TableGateway

```
use Laminas\Db\TableGateway\TableGateway;

$tableGateway = new TableGateway('posts', $adapter);
```

Create (Insert)

```
$tableGateway->insert([
   'title' => 'New Post',
   'content' => 'This is a test',
]);
```

Read (Select)

```
$rows = $tableGateway->select();
foreach ($rows as $row) {
   echo $row->title;
}
```

Update

```
$tableGateway->update(
   ['title' => 'Updated Title'],
   ['id' => 1]
);
```

Delete

```
$tableGateway->delete(['id' => 1]);
```

Project 2: Mini-Blog database-driven CRUD app + forms with Fieldset + CSRF + PostTable CRUD logic

Step 1: Create a PostFieldset
This will group our form elements for Post (reusable in Add/Edit forms).

module/Blog/src/Form/PostFieldset.php

```
namespace Blog\Form;

use Laminas\Form\Fieldset;
use Laminas\Form\Element;
use Laminas\Hydrator\ClassMethodsHydrator;
use Blog\Model\Post;

class PostFieldset extends Fieldset
{
   public function __construct()
   {
      parent::__construct('post');

      // Hydrator to map form <-> Post entity
      $this->setHydrator(new ClassMethodsHydrator());
      $this->setObject(new Post());

      $this->add([
         'name' => 'id',
         'type' => Element\Hidden::class,
      ]);

      $this->add([
         'name' => 'title',
         'type' => Element\Text::class,
```

```
            'options' => ['label' => 'Title'],
            'attributes' => ['required' => true, 'placeholder' => 'Enter blog title'],
        ]);

        $this->add([
            'name' => 'content',
            'type' => Element\Textarea::class,
            'options' => ['label' => 'Content'],
            'attributes' => ['required' => true, 'rows' => 5],
        ]);
    }
}
```

Step 2: Create a PostForm with CSRF
This form wraps the PostFieldset and adds a CSRF token for security.

module/Blog/src/Form/PostForm.php

```
namespace Blog\Form;

use Laminas\Form\Form;
use Laminas\Form\Element;

class PostForm extends Form
{
    public function __construct()
    {
        parent::__construct('post-form');

        // Use POST
        $this->setAttribute('method', 'post');

        // Add Fieldset
        $this->add([
            'type' => PostFieldset::class,
            'name' => 'post',
        ]);

        // Add CSRF protection
        $this->add([
            'type' => Element\Csrf::class,
            'name' => 'csrf',
            'options' => [
                'csrf_options' => ['timeout' => 600], // 10 min
            ],
        ]);

        // Submit button
        $this->add([
            'name' => 'submit',
            'type' => Element\Submit::class,
```

```
            'attributes' => ['value' => 'Save Post'],
        ]);
    }
}
```

Step 3: Controller Integration
Update BlogController so Add/Edit actions use PostForm.

module/Blog/src/Controller/BlogController.php

```
use Blog\Form\PostForm;
use Blog\Model\Post;

public function addAction()
{
    $form = new PostForm();
    $post = new Post();
    $form->bind($post);

    $request = $this->getRequest();
    if ($request->isPost()) {
        $form->setData($request->getPost());

        if ($form->isValid()) {
            $this->postTable->savePost($post); // Entity auto-filled
            $this->flashMessenger()->addSuccessMessage("Post created successfully!");
            return $this->redirect()->toRoute('blog');
        }
    }

    return ['form' => $form];
}

public function editAction()
{
    $id = (int) $this->params()->fromRoute('id', 0);
    if ($id === 0) {
        return $this->redirect()->toRoute('blog', ['action' => 'add']);
    }

    try {
        $post = $this->postTable->getPost($id);
    } catch (\Exception $e) {
        return $this->redirect()->toRoute('blog');
    }

    $form = new PostForm();
    $form->bind($post);

    $request = $this->getRequest();
    if ($request->isPost()) {
```

```
    $form->setData($request->getPost());

    if ($form->isValid()) {
      $this->postTable->savePost($post);
      $this->flashMessenger()->addSuccessMessage("Post updated successfully!");
      return $this->redirect()->toRoute('blog');
    }
  }

  return ['form' => $form, 'id' => $id];
}
```

Step 4: Views with CSRF

add.phtml

```
<h2>Add New Post</h2>
<?= $this->form()->openTag($form) ?>
  <?= $this->formCollection($form) ?>
<?= $this->form()->closeTag() ?>
```

edit.phtml

```
<h2>Edit Post</h2>
<?= $this->form()->openTag($form) ?>
  <?= $this->formCollection($form) ?>
<?= $this->form()->closeTag() ?>
```

Note: formCollection() renders fieldset (title, content, csrf, submit) automatically.

Step 5: Database CRUD (already wired)

We already built PostTable with savePost(), getPost(), fetchAll(), and deletePost(). With binding + hydrator, $post objects automatically populate from form data.

Step 6: Flash Messages (Optional UI Enhancement)

In layout.phtml, add:

```
<?php if ($this->flashMessenger()->hasSuccessMessages()): ?>
  <div class="alert alert-success">
    <?php foreach ($this->flashMessenger()->getSuccessMessages() as $msg): ?>
      <p><?= $msg ?></p>
    <?php endforeach; ?>
  </div>
<?php endif; ?>
```

Final Result

- ✓ Add Post → /blog/add shows a secure form (PostFieldset + CSRF).
- ✓ Edit Post → /blog/edit/:id pre-fills form with entity data.
- ✓ CSRF protection ensures secure submissions.
- ✓ PostTable CRUD saves to DB.
- ✓ Flash messages confirm success.

Laminas Components and Modules

Using various Laminas components

Laminas is component-based: you can use any component standalone (like laminas-log, laminas-cache) without the full MVC.

Using Laminas\Log

```
$logger = new Laminas\Log\Logger();
$logger->addWriter(new Laminas\Log\Writer\Stream('php://output'));
$logger->info("Hello, Laminas!");
```

Using Laminas\Cache

```
use Laminas\Cache\StorageFactory;
$cache = StorageFactory::factory(['adapter' => 'filesystem']);
$cache->setItem('foo', 'bar');
echo $cache->getItem('foo'); // bar
```

Using Laminas\Mail

```
use Laminas\Mail\Message;
$message = new Message();
$message->addTo('user@example.com')
    ->addFrom('admin@example.com')
    ->setSubject('Test')
    ->setBody('Hello world');
```

Using Laminas\Validator

```
use Laminas\Validator\EmailAddress;
$validator = new EmailAddress();
var_dump($validator->isValid("test@example.com")); // true
```

Creating and managing modules

Modules organize application features into self-contained units.

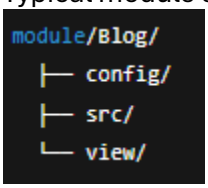
Create a module

```
php vendor/bin/laminas module:create Blog
```

Enable in modules.config.php

```
return [
  'Application',
  'Blog',
];
```

Typical module structure



Register services in module.config.php

```
'service_manager' => [
   'factories' => [
      Blog\Service\PostService::class => Blog\Factory\PostServiceFactory::class,
   ],
],
```

Error Handling and Logging

Debugging in Laminas

Debugging is done with built-in PHP tools, Laminas-specific modules, or third-party tools.

Enable error display in public/index.php

```
ini_set('display_errors', 1);
error_reporting(E_ALL);
```

Dump variables

```
var_dump($variable);
```

Use Laminas\Debug

```
use Laminas\Debug\Debug;
Debug::dump($variable);
```

Integrate Whoops (3rd party)

```
composer require filp/whoops
```

Configuring logging in Laminas

Logging helps trace errors, warnings, and app events.

Basic File Logger

```
use Laminas\Log\Logger;
use Laminas\Log\Writer\Stream;

$logger = new Logger();
$logger->addWriter(new Stream('data/logs/app.log'));
$logger->info("Application started");
```

Multiple Writers (File + Syslog)

```
$logger->addWriter(new Stream('data/logs/app.log'));
$logger->addWriter(new Laminas\Log\Writer\Syslog());
```

Log an Exception

```
try {
  throw new \Exception("Oops");
} catch (\Exception $e) {
  $logger->err($e->getMessage());
}
```

Log to console

```
$logger->addWriter(new Stream('php://output'));
```

Building REST APIs with Laminas

Creating RESTful APIs

Laminas makes it easy to build REST endpoints using controllers and JsonModel.

Define API route

```
'router' => [
 'routes' => [
  'api-posts' => [
   'type' => 'Segment',
   'options' => [
    'route' => '/api/posts[/:id]',
    'defaults' => [
     'controller' => Blog\Controller\Api\PostController::class,
    ],
   ],
  ],
 ],
],
```

API Controller

```
use Laminas\Mvc\Controller\AbstractRestfulController;
use Laminas\View\Model\JsonModel;

class PostController extends AbstractRestfulController {
  public function getList() {
    return new JsonModel(['posts' => [['id' => 1, 'title' => 'Hello']]]);
  }
  public function get($id) {
    return new JsonModel(['id' => $id, 'title' => 'Post ' . $id]);
  }
}
```

Access API

```
curl http://localhost:8080/api/posts
```

Handling JSON responses and API authentication

APIs usually respond with JSON and need authentication (e.g., API keys, JWT).

Return JSON Response

```
return new JsonModel(['status' => 'ok', 'data' => [1,2,3]]);
```

Custom HTTP status

```
$response = $this->getResponse();
$response->setStatusCode(201);
return new JsonModel(['message' => 'Created']);
```

Simple API Key check

```
$apiKey = $this->params()->fromHeader('X-API-KEY');
if ($apiKey !== 'my-secret-key') {
  return new JsonModel(['error' => 'Unauthorized']);
}
```

JWT Authentication (using firebase/php-jwt)

```
use Firebase\JWT\JWT;

$token = JWT::encode(['user' => 'john'], 'secret-key', 'HS256');
$decoded = JWT::decode($token, new \Firebase\JWT\Key('secret-key', 'HS256'));
```

Project 3: Mini REST API in Laminas CRUD operations + JWT authentication

Step 1: Install Required Packages

We'll need Laminas MVC (already installed if you followed our blog project) and a JWT library:

```
composer require firebase/php-jwt
```

Step 2: Define API Routes

Add this in module/Blog/config/module.config.php:

```
'router' => [
  'routes' => [
    'api-posts' => [
      'type' => 'Segment',
      'options' => [
        'route' => '/api/posts[/:id]',
        'defaults' => [
          'controller' => Blog\Controller\Api\PostController::class,
        ],
      ],
    ],
    'api-auth' => [
      'type' => 'Literal',
      'options' => [
```

```
            'route' => '/api/auth',
            'defaults' => [
              'controller' => Blog\Controller\Api\AuthController::class,
              'action' => 'login',
            ],
          ],
        ],
      ],
    ],
];
```

Step 3: Create the API Controller for Posts

module/Blog/src/Controller/Api/PostController.php

```
namespace Blog\Controller\Api;

use Laminas\Mvc\Controller\AbstractRestfulController;
use Laminas\View\Model\JsonModel;
use Blog\Model\PostTable;
use Blog\Model\Post;
use Firebase\JWT\JWT;
use Firebase\JWT\Key;

class PostController extends AbstractRestfulController
{
  private $postTable;
  private $jwtSecret = 'my-secret-key';

  public function __construct(PostTable $postTable)
  {
    $this->postTable = $postTable;
  }

  private function checkAuth()
  {
    $authHeader = $this->getRequest()->getHeader('Authorization');
    if (!$authHeader) {
      return false;
    }

    $token = str_replace('Bearer ', '', $authHeader->getFieldValue());

    try {
      $decoded = JWT::decode($token, new Key($this->jwtSecret, 'HS256'));
      return $decoded;
    } catch (\Exception $e) {
      return false;
    }
  }

  public function getList()
```

```
{
    if (!$this->checkAuth()) {
        return $this->unauthorizedResponse();
    }

    $posts = [];
    foreach ($this->postTable->fetchAll() as $post) {
        $posts[] = ['id' => $post->id, 'title' => $post->title, 'content' => $post->content];
    }

    return new JsonModel(['posts' => $posts]);
}

public function get($id)
{
    if (!$this->checkAuth()) {
        return $this->unauthorizedResponse();
    }

    try {
        $post = $this->postTable->getPost($id);
    } catch (\Exception $e) {
        return new JsonModel(['error' => 'Post not found']);
    }

    return new JsonModel(['id' => $post->id, 'title' => $post->title, 'content' => $post->content]);
}

public function create($data)
{
    if (!$this->checkAuth()) {
        return $this->unauthorizedResponse();
    }

    $post = new Post();
    $post->exchangeArray($data);
    $this->postTable->savePost($post);

    return new JsonModel(['message' => 'Post created', 'id' => $post->id]);
}

public function update($id, $data)
{
    if (!$this->checkAuth()) {
        return $this->unauthorizedResponse();
    }

    $post = $this->postTable->getPost($id);
    $post->exchangeArray(array_merge((array)$post, $data));
    $this->postTable->savePost($post);
```

```
        return new JsonModel(['message' => 'Post updated']);
    }

    public function delete($id)
    {
        if (!$this->checkAuth()) {
            return $this->unauthorizedResponse();
        }

        $this->postTable->deletePost($id);
        return new JsonModel(['message' => 'Post deleted']);
    }

    private function unauthorizedResponse()
    {
        $response = $this->getResponse();
        $response->setStatusCode(401);
        return new JsonModel(['error' => 'Unauthorized']);
    }
}
```

Step 4: Create Authentication Controller

module/Blog/src/Controller/Api/AuthController.php

```
namespace Blog\Controller\Api;

use Laminas\Mvc\Controller\AbstractActionController;
use Laminas\View\Model\JsonModel;
use Firebase\JWT\JWT;

class AuthController extends AbstractActionController
{
    private $jwtSecret = 'my-secret-key';

    public function loginAction()
    {
        $request = $this->getRequest();

        if ($request->isPost()) {
            $data = json_decode($request->getContent(), true);

            $username = $data['username'] ?? '';
            $password = $data['password'] ?? '';

            // Hardcoded credentials (in real app, query DB)
            if ($username === 'admin' && $password === 'password') {
                $payload = [
                    'sub' => $username,
                    'iat' => time(),
```

```
            'exp' => time() + 3600 // 1 hour expiry
          ];
          $jwt = JWT::encode($payload, $this->jwtSecret, 'HS256');

          return new JsonModel(['token' => $jwt]);
        }
      }

      $response = $this->getResponse();
      $response->setStatusCode(401);
      return new JsonModel(['error' => 'Invalid credentials']);
    }
}
```

Step 5: Configure Controller Factories

module/Blog/config/module.config.php

```
'controllers' => [
    'factories' => [
        Blog\Controller\Api\PostController::class => function($container) {
            return new Blog\Controller\Api\PostController(
                $container->get(Blog\Model\PostTable::class)
            );
        },
        Blog\Controller\Api\AuthController::class => Laminas\ServiceManager\Factory\InvokableFactory::class,
    ],
],
```

Step 6: Test the API

Get Token

```
curl -X POST http://localhost:8080/api/auth \
  -H "Content-Type: application/json" \
  -d '{"username":"admin","password":"password"}'
```

Response:

```
{"token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..."}
```

Use Token for Requests

```
curl -X GET http://localhost:8080/api/posts \
  -H "Authorization: Bearer <token>"
```

Response:

```
{"posts":[{"id":1,"title":"Hello","content":"First post"}]}
```

Create Post

```
curl -X POST http://localhost:8080/api/posts \
  -H "Authorization: Bearer <token>" \
  -H "Content-Type: application/json" \
```

```
 -d '{"title":"API Post","content":"Created via API"}'
```

Update Post

```
curl -X PUT http://localhost:8080/api/posts/1 \
 -H "Authorization: Bearer <token>" \
 -H "Content-Type: application/json" \
 -d '{"title":"Updated Title"}'
```

Delete Post

```
curl -X DELETE http://localhost:8080/api/posts/1 \
 -H "Authorization: Bearer <token>"
```

Summary

- /api/auth → login & return JWT
- /api/posts → CRUD with JWT protection
- JSON responses + HTTP status codes
- JWT authentication middleware built directly into the controller

Performance Optimization and Security

Best practices for improving performance

Performance tuning ensures that Laminas applications scale under load.

Use Configuration Caching
Laminas can cache its merged configuration to avoid re-parsing PHP arrays every request.

```
// config/modules.config.php
'config_cache_enabled' => true,
'config_cache_key'    => 'app_config',
'cache_dir'          => 'data/cache/',
```

*First request builds cache, subsequent requests read from it.

Enable Module Class Map Autoloading
Instead of scanning file paths, Laminas can autoload from prebuilt maps.

```
php vendor/bin/laminas classmap:generate module/Application/src >
module/Application/autoload_classmap.php
```

*Faster class resolution.

Optimize View Rendering
Use .phtml partials and avoid excessive loops in templates.

Example:

```
<?= $this->partial('partials/post', ['post' => $post]); ?>
```

*Improves reusability and performance.

Use Database Connection Pooling (PDO persistent connections)

```
'db' => [
   'driver' => 'Pdo',
   'dsn' => 'mysql:dbname=laminas_blog;host=localhost',
   'driver_options' => [
      PDO::ATTR_PERSISTENT => true,
   ],
],
```

*Reduces connection overhead.

Leverage Caching (Laminas\Cache)

```
use Laminas\Cache\StorageFactory;
$cache = StorageFactory::factory(['adapter' => 'apc']);
$cache->setItem('homepage_data', $data);
```

*Avoid recomputing or requerying expensive data.

Securing a Laminas application (User authentication, Role-based access control (RBAC))

Security includes authentication (verifying who you are) and authorization (what you can do). Laminas provides tools like Laminas\Authentication and Laminas\Permissions\Rbac.

1. User Authentication

Laminas uses AuthenticationService to validate users.

```
Setup AuthenticationService

use Laminas\Authentication\AuthenticationService;
$authService = new AuthenticationService();
```

*Central object to handle login/logout.

Login with Adapter (DB table)

```
use Laminas\Authentication\Adapter\DbTable\CallbackCheckAdapter;

$adapter = new CallbackCheckAdapter(
   $dbAdapter,
   'users',    // table
   'username',  // identity column
   'password',  // credential column
   function($hash, $password) { return password_verify($password, $hash); }
);

$adapter->setIdentity($username)->setCredential($password);
$result = $authService->authenticate($adapter);
```

Check Authentication

```
if ($authService->hasIdentity()) {
   echo "Logged in as " . $authService->getIdentity();
}
```

Logout

```
$authService->clearIdentity();
```

2. Role-Based Access Control (RBAC)

RBAC defines roles (e.g., admin, editor, user) and permissions (e.g., manage posts, view posts).

Create RBAC Instance

```
use Laminas\Permissions\Rbac\Rbac;
$rbac = new Rbac();
```

Add Roles with Permissions

```
$admin = $rbac->addRole('admin');
$admin->addPermission('manage_users');
$admin->addPermission('manage_posts');

$editor = $rbac->addRole('editor');
$editor->addPermission('edit_posts');
```

Check Access

```
if ($rbac->isGranted('admin', 'manage_posts')) {
   echo "Admin can manage posts.";
}
```

Nested Roles (Inheritance)

```
$editor = $rbac->addRole('editor', 'user'); // inherits user
```

Integrating RBAC with Controllers

```
if (!$rbac->isGranted($userRole, 'edit_posts')) {
   return $this->redirect()->toRoute('forbidden');
}
```

Project 4: Mini-Blog with User Login and RBAC

Step 1: Create Users Table

Add a users table to the database:

```
CREATE TABLE users (
   id INT AUTO_INCREMENT PRIMARY KEY,
   username VARCHAR(50) NOT NULL UNIQUE,
   password VARCHAR(255) NOT NULL,
   role ENUM('admin','editor','user') DEFAULT 'user'
);
```

```
-- Add a default admin
INSERT INTO users (username, password, role)
VALUES ('admin', PASSWORD('secret'), 'admin');
```

*In modern PHP, store with password_hash() not PASSWORD().

Example for seed script:

```
$hash = password_hash('secret', PASSWORD_DEFAULT);
```

Step 2: Configure AuthenticationService

Add authentication factory in module/Blog/config/module.config.php:

```
use Laminas\Authentication\AuthenticationService;
use Laminas\Authentication\Adapter\DbTable\CallbackCheckAdapter;

'service_manager' => [
  'factories' => [
    AuthenticationService::class => function($container) {
      $dbAdapter = $container->get(\Laminas\Db\Adapter\Adapter::class);

      $adapter = new CallbackCheckAdapter(
        $dbAdapter,
        'users',       // table
        'username',    // identity column
        'password',    // credential column
        function($hash, $password) {
          return password_verify($password, $hash);
        }
      );

      $authService = new AuthenticationService();
      $authService->setAdapter($adapter);
      return $authService;
    },
  ],
],
```

Step 3: Create AuthController for Login/Logout

module/Blog/src/Controller/AuthController.php

```
namespace Blog\Controller;

use Laminas\Mvc\Controller\AbstractActionController;
use Laminas\View\Model\ViewModel;
use Laminas\Authentication\AuthenticationService;

class AuthController extends AbstractActionController
{
```

```
    private $authService;

    public function __construct(AuthenticationService $authService)
    {
        $this->authService = $authService;
    }

    public function loginAction()
    {
        $request = $this->getRequest();

        if ($request->isPost()) {
            $data = $request->getPost();
            $adapter = $this->authService->getAdapter();
            $adapter->setIdentity($data['username']);
            $adapter->setCredential($data['password']);

            $result = $this->authService->authenticate();

            if ($result->isValid()) {
                $this->authService->getStorage()->write($adapter->getResultRowObject(null, 'password'));
                $this->flashMessenger()->addSuccessMessage("Login successful!");
                return $this->redirect()->toRoute('blog');
            } else {
                $this->flashMessenger()->addErrorMessage("Invalid credentials.");
            }
        }

        return new ViewModel();
    }

    public function logoutAction()
    {
        $this->authService->clearIdentity();
        $this->flashMessenger()->addSuccessMessage("Logged out.");
        return $this->redirect()->toRoute('login');
    }
}
```

Step 4: Add Login Form

module/Blog/view/blog/auth/login.phtml

```
<h2>Login</h2>
<?php if ($this->flashMessenger()->hasMessages()): ?>
    <?php foreach ($this->flashMessenger()->getMessages() as $msg): ?>
        <p><?= $msg ?></p>
    <?php endforeach; ?>
<?php endif; ?>

<form method="post">
```

```
  <label>Username: <input type="text" name="username"></label><br>
  <label>Password: <input type="password" name="password"></label><br>
  <button type="submit">Login</button>
</form>
```

Step 5: Add RBAC Service

module/Blog/src/Service/RbacService.php

```
namespace Blog\Service;

use Laminas\Permissions\Rbac\Rbac;

class RbacService
{
  private $rbac;

  public function __construct()
  {
    $this->rbac = new Rbac();

    // Define roles & permissions
    $admin = $this->rbac->addRole('admin');
    $admin->addPermission('post.manage');

    $editor = $this->rbac->addRole('editor', 'user');
    $editor->addPermission('post.edit');

    $user = $this->rbac->addRole('user');
    $user->addPermission('post.view');
  }

  public function isGranted($role, $permission)
  {
    return $this->rbac->isGranted($role, $permission);
  }
}
```

Register in module.config.php:

```
'service_manager' => [
  'factories' => [
    Blog\Service\RbacService::class => Laminas\ServiceManager\Factory\InvokableFactory::class,
  ],
],
```

Step 6: Protect BlogController with RBAC

inside BlogController.php

```
use Blog\Service\RbacService;
use Laminas\Authentication\AuthenticationService;
```

```
private $authService;
private $rbacService;

public function __construct(PostTable $postTable, AuthenticationService $authService, RbacService
$rbacService)
{
   $this->postTable = $postTable;
   $this->authService = $authService;
   $this->rbacService = $rbacService;
}

private function checkAccess($permission)
{
   if (!$this->authService->hasIdentity()) {
      return false;
   }
   $user = $this->authService->getIdentity();
   return $this->rbacService->isGranted($user->role, $permission);
}

public function addAction()
{
   if (!$this->checkAccess('post.manage')) {
      return $this->redirect()->toRoute('login');
   }
   // ... add post logic
}
```

Step 7: Wiring Controllers with Factories

module/Blog/config/module.config.php

```
'controllers' => [
   'factories' => [
      Blog\Controller\AuthController::class => function($c) {
         return new Blog\Controller\AuthController($c->get(\Laminas\Authentication\AuthenticationService::class));
      },
      Blog\Controller\BlogController::class => function($c) {
         return new Blog\Controller\BlogController(
            $c->get(Blog\Model\PostTable::class),
            $c->get(\Laminas\Authentication\AuthenticationService::class),
            $c->get(Blog\Service\RbacService::class)
         );
      },
   ],
],
```

Step 8: Define Routes

module.config.php

```
'router' => [
  'routes' => [
    'login' => [
      'type' => 'Literal',
      'options' => [
        'route' => '/login',
        'defaults' => [
          'controller' => Blog\Controller\AuthController::class,
          'action'    => 'login',
        ],
      ],
    ],
    'logout' => [
      'type' => 'Literal',
      'options' => [
        'route' => '/logout',
        'defaults' => [
          'controller' => Blog\Controller\AuthController::class,
          'action'    => 'logout',
        ],
      ],
    ],
  ],
],
```

Final Behavior

1. Visit /login → Enter username/password.
2. If valid → User stored in session (AuthenticationService).
3. RBAC checks:
    a. admin → can add, edit, delete posts.
    b. editor → can edit but not delete.
    c. user → can only view posts.
4. Logout at /logout.

Now our Blog app has:

- ✓ Authentication (login/logout)
- ✓ Role-based authorization (RBAC)
- ✓ Protected routes (add/edit/delete only for admins/editors)