API with CodeIgniter4 and Containerization with Docker

Contents

Understanding RESTful APIs

Introduction to RESTful architecture

REST (Representational State Transfer) is an architectural style for designing networked applications. It relies on a stateless, client-server communication model, often over HTTP. Key aspects include:

- Stateless Communication: Each request from a client to a server must contain all the information needed to understand and complete the request. The server doesn't store client state.
- Resource-Oriented: RESTful services focus on resources, identifiable by URIs (Uniform Resource Identifiers).
- Uniform Interface: RESTful APIs use consistent conventions and methods across the API, making them easier to understand and work with.

HTTP methods (GET, POST, PUT, PATCH, DELETE)

HTTP methods define the operations you can perform on a resource. Here's how each is typically used:

- GET: Retrieve data from the server. It's safe and idempotent (repeated calls don't change the resource).
- POST: Create a new resource. This method is not idempotent, as calling it multiple times can create multiple resources.
- PUT: Update a resource by replacing it entirely. It's idempotent.
- PATCH: Partially update a resource, only changing the specified fields.
- DELETE: Remove a resource from the server.

RESTful API design principles

RESTful APIs are based on certain design principles to ensure consistency and usability:

- ✓ Statelessness: Ensures that each request is independent and does not rely on previous requests.
- ✓ Client-Server Architecture: Separates user interface from data storage, allowing each to evolve independently.
- ✓ Cacheability: Responses must explicitly indicate if they're cacheable or not to improve performance.
- ✓ Layered System: Clients cannot tell if they're directly connected to the end server, allowing for intermediary servers to improve scalability.

Common API design patterns

To enhance the functionality and usability of RESTful APIs, these design patterns are commonly used:

- ✓ Noun-based URIs: Resources are defined as nouns (e.g., /users, /orders) rather than verbs.
- ✓ Hierarchical URIs: Resource relationships are shown in URIs, like /users/{user_id}/orders/{order_id}.
- ✓ Versioning: API versions should be explicitly included in the URI (e.g., /v1/users).
- ✓ Error Handling: Use standard HTTP status codes (e.g., 404 for "Not Found", 500 for server errors) with detailed messages.

Introduction to CodeIgniter 4

Overview of CodeIgniter 4

CodeIgniter 4 is a PHP-based MVC framework known for its lightweight and straightforward approach. It's designed to simplify the development of web applications by providing a clear structure and efficient tools. Key features include:

- ✓ MVC Architecture: Separation of logic, presentation, and data in applications.
- ✓ Lightweight: Suitable for small-to-medium applications with low server overhead.
- ✓ Extensive Libraries: CodeIgniter provides libraries to handle tasks like sessions, file uploads, and form validation.

Installation and setup

Step-by-Step Installation

1. System Requirements: Ensure you have PHP 7.2 (or higher) and Composer installed.
2. Download CodeIgniter: Open your terminal and navigate to the folder where you want to set up the project. Use Composer to install CodeIgniter 4 by running:

```
composer create-project codeigniter4/appstarter myproject
```

*This will create a folder named myproject with all CodeIgniter files.

3. Configuration:

Go to the myproject directory:

```
cd myproject
```

Open the .env file and rename it from env to .env.

```
Set CI_ENVIRONMENT = development in .env to enable development mode for better debugging.
```

Run the Application: Start the server to ensure everything is set up properly:

```
php spark serve
```

*Access the project at http://localhost:8080 to see the default CodeIgniter welcome page.

Project structure and MVC architecture

CodeIgniter's folder structure and how it fits into the MVC (Model-View-Controller) framework:

- app: Contains all application code.
  - Controllers: Code for request handling logic.
  - Models: Handles data logic, typically communicating with the database.
  - Views: Contains the HTML, CSS, and JavaScript for presenting data to users.
- public: Contains assets like images, JavaScript, and CSS files.
- writable: Stores cache, session data, and logs.

In CodeIgniter's MVC:

- Controllers manage incoming requests and communicate with Models.
- Models retrieve data from the database.
- Views display data and the user interface.

Routing in CodeIgniter 4

Routing defines how URLs map to controllers and actions.

Step-by-Step Demonstration of Routing

1. Open app/Config/Routes.php.
2. Define a new route to point to a controller and a method:

```
$routes->get('hello', 'HelloController::index');
```

*This means when users access http://localhost:8080/hello, it will route to the index method of HelloController.

3. Custom Route: You can create routes with parameters as well:

```
$routes->get('user/(:num)', 'UserController::profile/$1');
```

*Here, (:num) allows only numbers in the URL, and $1 passes this number to the profile method of UserController.

Controllers and views

1. Create a Controller:
   In the app/Controllers folder, create a file named HelloController.php with the following content:

   HelloController.php

```
<?php

namespace App\Controllers;

class HelloController extends BaseController
{
   public function index()
   {
      return view('hello');
   }
}
```

   *This controller has an index method that loads a view named hello.

2. Create a View:
   In the app/Views folder, create a file named hello.php.

   hello.php

```
<h1>Hello, CodeIgniter 4!</h1>
<p>Welcome to your first view.</p>
```

   Access the View:

```
http://localhost:8080/hello
```

Example 1: Static Pages

1. Goal: Set up routes to serve static pages like "About" and "Contact."
2. Routing
   Open app/Config/Routes.php.

```
$routes->get('about', 'PagesController::about');
$routes->get('contact', 'PagesController::contact');
```

3. Controller
   In app/Controllers, create a new file called PagesController.php.

```
<?php
namespace App\Controllers;

class PagesController extends BaseController
{
  public function about()
  {
    return view('about');
  }

  public function contact()
  {
    return view('contact');
  }
}
```

4. Views
   In app/Views, create two files: about.php and contact.php.

   about.php:

```
<h1>About Us</h1>
<p>This is the about page.</p>
```

   contact.php:

```
<h1>Contact Us</h1>
<p>This is the contact page.</p>
```

   *Access these pages via http://localhost:8080/about and http://localhost:8080/contact.

Example 2: Passing Parameters to Controllers

1. Goal: Display a user profile by ID.
2. Routing
   In app/Config/Routes.php, add:

```
$routes->get(‘user/(:num)’, ‘UserController::profile/$1’);
```

3. Controller
   In app/Controllers, create UserController.php.
   Define the profile method to receive the user ID:

```
<?php
namespace App\Controllers;
```

```
class UserController extends BaseController
{
   public function profile($id)
   {
      return view('user_profile', ['id' => $id]);
   }
}
```

4. View
   In app/Views, create user_profile.php.
   Add content to display the user ID:

```
<h1>User Profile</h1>
<p>User ID: <?= esc($id) ?></p>
```

   *Access this page with http://localhost:8080/user/1 (or any other ID).

Example 3: Multiple Parameters in Routes

1. Goal: Show a blog post by category and post ID.
2. Routing
   In app/Config/Routes.php, add:

```
$routes->get('blog/(:any)/(:num)', 'BlogController::post/$1/$2');
```

3. Controller
   In app/Controllers, create BlogController.php.
   Define the post method to handle the category and ID:

```
<?php
namespace App\Controllers;

class BlogController extends BaseController
{
   public function post($category, $id)
   {
      return view('blog_post', ['category' => $category, 'id' => $id]);
   }
}
```

4. View
   In app/Views, create blog_post.php.
   Add content to display the category and post ID:

```
<h1>Blog Post</h1>
<p>Category: <?= esc($category) ?></p>
<p>Post ID: <?= esc($id) ?></p>
```

   *Access this page with http://localhost:8080/blog/technology/1.

Example 4: POST Request Handling

1. Goal: Process a contact form submission.
2. Routing

In app/Config/Routes.php, add:

```
$routes->get('contact', 'ContactController::form');
$routes->post('contact', 'ContactController::submit');
```

3. Controller
   In app/Controllers, create ContactController.php.
   Define form to show the form and submit to handle form data:

```
<?php
namespace App\Controllers;

class ContactController extends BaseController
{
  public function form()
  {
    return view('contact_form');
  }

  public function submit()
  {
    $name = $this->request->getPost('name');
    $email = $this->request->getPost('email');
    return view('contact_success', ['name' => $name, 'email' => $email]);
  }
}
```

4. Views
   In app/Views, create contact_form.php:

```
<form method="post" action="/contact">
  <input type="text" name="name" placeholder="Your Name">
  <input type="email" name="email" placeholder="Your Email">
  <button type="submit">Submit</button>
</form>
```

   Create contact_success.php to display the submitted data:

```
<h1>Thank You!</h1>
<p>Name: <?= esc($name) ?></p>
<p>Email: <?= esc($email) ?></p>
```

   *Access the form at http://localhost:8080/contact and submit it to see the success page.

Example 5: Redirecting Routes

1. Goal: Redirect the user from an old URL to a new one.
2. Routing
   In app/Config/Routes.php, use:

```
$routes->get('old-page', 'RedirectController::oldPage');
$routes->get('new-page', 'RedirectController::newPage');
```

3. Controller
   In app/Controllers, create RedirectController.php.
   Define the methods:

```php
<?php
namespace App\Controllers;

class RedirectController extends BaseController
{
  public function oldPage()
  {
    return redirect()->to('/new-page');
  }

  public function newPage()
  {
    return view('new_page');
  }
}
```

4. View
   In app/Views, create new_page.php:

```
<h1>New Page</h1>
<p>Welcome to the new page!</p>
```

*Access http://localhost:8080/old-page, and you will be redirected to /new-page.

Example 6: Route Grouping

1. Goal: Group related routes under a common prefix, such as for an admin dashboard.
2. Routing. In app/Config/Routes.php, define a route group with a prefix:

```
$routes->group('admin', function($routes) {
  $routes->get('dashboard', 'AdminController::dashboard');
  $routes->get('users', 'AdminController::users');
  $routes->get('settings', 'AdminController::settings');
});
```

3. Controller
   In app/Controllers, create AdminController.php. Define methods for each route:

```php
<?php
namespace App\Controllers;

class AdminController extends BaseController
{
  public function dashboard()
  {
    return view('admin/dashboard');
  }

  public function users()
  {
    return view('admin/users');
  }
```

```
    public function settings()
    {
        return view('admin/settings');
    }
}
```

4. Views
   In app/Views, create an admin folder and add three files: dashboard.php, users.php, and settings.php.

   dashboard.php:

```
<h1>Admin Dashboard</h1>
<p>Welcome to the dashboard.</p>
```

   users.php:

```
<h1>User Management</h1>
<p>Manage your users here.</p>
```

   settings.php:

```
<h1>Settings</h1>
<p>Manage application settings here.</p>
```

   *Access these pages at http://localhost:8080/admin/dashboard, http://localhost:8080/admin/users, and http://localhost:8080/admin/settings.

Example 7: Custom Error Page

1. Goal: Define a custom 404 error page for undefined routes.
2. Routing
   In app/Config/Routes.php, add a custom 404 override:

```
$routes->set404Override('ErrorController::show404');
```

3. Controller
   In app/Controllers, create ErrorController.php. Define the show404 method:

```
<?php
namespace App\Controllers;

class ErrorController extends BaseController
{
    public function show404()
    {
        return view('errors/custom_404');
    }
}
```

4. View
   In app/Views/errors, create custom_404.php:

```
<h1>404 - Page Not Found</h1>
<p>Sorry, the page you are looking for does not exist.</p>
```

   *Now, if a user visits an undefined route like http://localhost:8080/nonexistent, they will see your custom 404 page.

Example 8: RESTful Controller for CRUD Operations

1. Goal: Set up a RESTful controller to handle basic CRUD operations for a "Product" resource.
2. Routing
   In app/Config/Routes.php, define RESTful routes:

```
$routes->resource('product');
```

   *This will automatically create routes for index, show, create, update, and delete methods.

3. Controller
   In app/Controllers, create ProductController.php and extend ResourceController:

```
<?php
namespace App\Controllers;

use CodeIgniter\RESTful\ResourceController;

class ProductController extends ResourceController
{
   public function index()
   {
      return view('product/index');
   }

   public function show($id = null)
   {
      return view('product/show', ['id' => $id]);
   }

   public function create()
   {
      return view('product/create');
   }

   public function update($id = null)
   {
      return view('product/edit', ['id' => $id]);
   }

   public function delete($id = null)
   {
      return redirect()->to('/product');
   }
}
```

4. Views
   In app/Views/product, create four views: index.php, show.php, create.php, and edit.php.

   index.php:

```
<h1>All Products</h1>
```

show.php:

```
<h1>Product Details</h1>
<p>Product ID: <?= esc($id) ?></p>
```

create.php:

```
<h1>Create New Product</h1>
```

edit.php:

```
<h1>Edit Product</h1>
<p>Product ID: <?= esc($id) ?></p>
```

*Access http://localhost:8080/product to view these CRUD operations in action.

Example 9: Optional Route Parameters

1. Goal: Set up a route with an optional parameter.
2. Routing
   In app/Config/Routes.php, define a route with an optional id parameter:

```
$routes->get('post/(:num)?', 'PostController::view/$1');
```

3. Controller
   In app/Controllers, create PostController.php.
   Define the view method to handle cases with and without an ID:

```
<?php
namespace App\Controllers;

class PostController extends BaseController
{
   public function view($id = null)
   {
      if ($id) {
         return view('post/single', ['id' => $id]);
      }
      return view('post/all');
   }
}
```

4. Views
   In app/Views/post, create all.php and single.php.

all.php:

```
<h1>All Posts</h1>
<p>Listing all blog posts here.</p>
```

single.php:

```
<h1>Post Details</h1>
<p>Post ID: <?= esc($id) ?></p>
```

*Access http://localhost:8080/post to see all posts, or http://localhost:8080/post/1 to view a specific post.

Example 10: Route Constraints

1. Goal: Set up a route with constraints to allow only numeric IDs.
2. Routing
   In app/Config/Routes.php, define a route with constraints:

```
$routes->get('article/(:num)', 'ArticleController::view/$1', ['as' => 'article_view']);
```

3. Controller
   In app/Controllers, create ArticleController.php. Define the view method:

```
<?php
namespace App\Controllers;

class ArticleController extends BaseController
{
   public function view($id)
   {
      return view('article/view', ['id' => $id]);
   }
}
```

4. View
   In app/Views/article, create view.php:

```
<h1>Article Details</h1>
<p>Article ID: <?= esc($id) ?></p>
```

   *Access this with a numeric ID, e.g., http://localhost:8080/article/10.

Building Basic RESTful APIs with CodeIgniter 4

Creating basic CRUD (Create, Read, Update, Delete) operations

We'll create CRUD operations for a "Product" resource.

Step 1: Setting up the Database

Create a products table in your database with the following SQL:

```
CREATE TABLE products (
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(100),
   description TEXT,
   price DECIMAL(10,2)
);
```

Step 2: Configure Database Connection

Open app/Config/Database.php and set your database credentials.

Step 3: Create a Model

In app/Models, create ProductModel.php:

```
<?php
namespace App\Models;

use CodeIgniter\Model;
```

```
class ProductModel extends Model
{
   protected $table = 'products';
   protected $allowedFields = ['name', 'description', 'price'];
}
```

Step 4: Create a Controller

In app/Controllers, create ProductController.php:

```
<?php
namespace App\Controllers;

use App\Models\ProductModel;

class ProductController extends BaseController
{
   public function index()
   {
      $model = new ProductModel();
      $data['products'] = $model->findAll();
      return view('product/index', $data);
   }

   public function create()
   {
      return view('product/create');
   }

   public function store()
   {
      $model = new ProductModel();
      $model->save($this->request->getPost());
      return redirect()->to('/product');
   }

   public function edit($id)
   {
      $model = new ProductModel();
      $data['product'] = $model->find($id);
      return view('product/edit', $data);
   }

   public function update($id)
   {
      $model = new ProductModel();
      $model->update($id, $this->request->getPost());
      return redirect()->to('/product');
   }
```

```
    public function delete($id)
    {
        $model = new ProductModel();
        $model->delete($id);
        return redirect()->to('/product');
    }
}
```

Step 5: Create Views
In app/Views/product, create:

- index.php for listing products.
- create.php for adding a product.
- edit.php for editing a product.

You can now access CRUD functionality by navigating to routes like /product, /product/create, and /product/edit/{id}.

Handling HTTP requests and responses

Example 1: Retrieving Data with GET
Define a route:

```
$routes->get('product', 'ProductController::index');
```

Use GET to retrieve data:

```
$data = $this->request->getGet();
```

Example 2: Submitting Data with POST
Define a route:

```
$routes->post('product/store', 'ProductController::store');
```

Access POST data:

```
$name = $this->request->getPost('name');
```

Example 3: JSON Response

```
return $this->response->setJSON(['message' => 'Success']);
```

Example 4: Redirect

```
return redirect()->to('/product');
```

Example 5: Sending HTTP Status Codes

```
return $this->response->setStatusCode(404, 'Product Not Found');
```

Example 6: Handling Query Parameters in GET Requests
In CodeIgniter, you can capture query parameters from a URL, such as ?page=1&category=tech.

Define a Route:

```
$routes->get('articles', 'ArticleController::index');
```

Controller Method: In app/Controllers/ArticleController.php,
capture the query parameters using $this->request->getGet():

```
<?php
namespace App\Controllers;

class ArticleController extends BaseController
{
   public function index()
   {
      $page = $this->request->getGet('page'); // e.g., 1
      $category = $this->request->getGet('category'); // e.g., tech

      return view('articles/index', ['page' => $page, 'category' => $category]);
   }
}
```

View: Display the parameters in app/Views/articles/index.php:

```
<h1>Articles</h1>
<p>Page: <?= esc($page) ?></p>
<p>Category: <?= esc($category) ?></p>
```

*Access it with http://localhost:8080/articles?page=1&category=tech.

Example 7: Handling File Uploads in POST Requests

Define a Route:

```
$routes->post('upload', 'FileController::upload');
```

Controller Method: In app/Controllers/FileController.php,
handle file upload using $this->request->getFile():

```
<?php
namespace App\Controllers;

class FileController extends BaseController
{
   public function upload()
   {
      $file = $this->request->getFile('userfile');

      if ($file->isValid() && !$file->hasMoved()) {
         $file->move(WRITEPATH . 'uploads');
         return $this->response->setJSON(['message' => 'File uploaded successfully']);
      } else {
         return $this->response->setStatusCode(400, 'Failed to upload file');
      }
   }
}
```

HTML Form: In a view (e.g., app/Views/upload.php), add a form to upload files:

```
<form method="post" action="/upload" enctype="multipart/form-data">
  <input type="file" name="userfile">
  <button type="submit">Upload</button>
</form>
```

*Access the form at http://localhost:8080/upload.

Example 8: Handling PUT Requests for Resource Updates

To handle a PUT request (often used for updates), use CodeIgniter's put() method.

Define a Route:

```
$routes->put('product/update/(:num)', 'ProductController::update/$1');
```

Controller Method: In app/Controllers/ProductController.php,
use $this->request->getRawInput() for data in the PUT request:

```
<?php
namespace App\Controllers;

use App\Models\ProductModel;

class ProductController extends BaseController
{
  public function update($id)
  {
    $model = new ProductModel();
    $data = $this->request->getRawInput();

    if ($model->update($id, $data)) {
      return $this->response->setJSON(['message' => 'Product updated successfully']);
    } else {
      return $this->response->setStatusCode(400, 'Failed to update product');
    }
  }
}
```

*This handles JSON data sent with a PUT request for updating a product.

Example 9: Custom HTTP Headers

You can handle and send custom headers in the response to control caching, security, and content type.

Define a Route:

```
$routes->get('custom-headers', 'HeaderController::index');
```

Controller Method: In app/Controllers/HeaderController.php, add custom headers to the response:

```
<?php
namespace App\Controllers;

class HeaderController extends BaseController
{
```

```
    public function index()
    {
        $this->response->setHeader('Cache-Control', 'no-store, no-cache, must-revalidate');
        $this->response->setHeader('X-Content-Type-Options', 'nosniff');
        $this->response->setHeader('X-Frame-Options', 'DENY');
        return $this->response->setJSON(['message' => 'Custom headers added']);
    }
}
```

*Access this at http://localhost:8080/custom-headers and inspect the headers in your browser's network tools.

Example 10: Handling JSON Requests and Responses

CodeIgniter allows handling JSON data in requests, often used with REST APIs.

Define a Route:

```
$routes->post('api/product', 'ApiController::createProduct');
```

Controller Method: In app/Controllers/ApiController.php, capture JSON input with $this->request->getJSON():

```
<?php
namespace App\Controllers;

use App\Models\ProductModel;

class ApiController extends BaseController
{
    public function createProduct()
    {
        $model = new ProductModel();
        $json = $this->request->getJSON();

        $data = [
            'name' => $json->name,
            'description' => $json->description,
            'price' => $json->price,
        ];

        if ($model->save($data)) {
            return $this->response->setStatusCode(201)->setJSON(['message' => 'Product created']);
        } else {
            return $this->response->setStatusCode(400)->setJSON(['message' => 'Failed to create product']);
        }
    }
}
```

*This allows for handling JSON data in the request and sending JSON responses.

Implementing data validation and error handling

Step 1: Define Validation Rules

In app/Controllers/ProductController.php, use:

```
$rules = [
   'name' => 'required|min_length[3]',
   'price' => 'required|decimal',
];
```

Step 2: Apply Validation

Add to store and update methods:

```
if (!$this->validate($rules)) {
   return view('product/create', [
      'validation' => $this->validator
   ]);
}
```

Step 3: Display Errors in the View

In create.php, show errors:

```
<?php if (isset($validation)): ?>
   <div><?= $validation->listErrors() ?></div>
<?php endif; ?>
```

Step 4: Custom Error Messages

Customize in app/Language/en/Validation.php.

List of validation rules and what each does:

1. Basic Validation Rules
   - required: The field must not be empty.
   - matches[field_name]: Ensures that the input value matches the value of another field (e.g., matches[password_confirm]).
   - differs[field_name]: Ensures that the input value differs from the value of another field.
   - is_unique[table.field]: Ensures the value is unique in a specified database table and field (e.g., is_unique[users.email]).
   - min_length[length]: Ensures the field is at least a specified number of characters long.
   - max_length[length]: Ensures the field does not exceed a specified number of characters.
   - exact_length[length]: Ensures the field is exactly a specified number of characters.
   - in_list[item1,item2,...]: Ensures the value is one of the items in a specified list.
   - alpha: Allows only alphabetic characters (a–z, A–Z).
   - alpha_numeric: Allows only alphabetic and numeric characters.
   - alpha_numeric_space: Allows only alphabetic, numeric, and space characters.
   - alpha_dash: Allows only alphabetic characters, underscores, and dashes.
   - alpha_numeric_punct: Allows alphanumeric characters and commonly used punctuation (e.g., ~ ! # $ % & * - _ + = | : .).
   - alpha_space: Allows only alphabetic characters and spaces.

2. Numeric Validation Rules
   - numeric: Ensures the field contains only numeric characters.
   - integer: Ensures the field contains only integers.

- decimal: Ensures the field is a decimal number.
- is_natural: Ensures the field contains only natural numbers (0, 1, 2, ...).
- is_natural_no_zero: Ensures the field contains only natural numbers greater than zero (1, 2, 3, ...).
- greater_than[value]: Ensures the field contains a number greater than the specified value.
- greater_than_equal_to[value]: Ensures the field contains a number greater than or equal to the specified value.
- less_than[value]: Ensures the field contains a number less than the specified value.
- less_than_equal_to[value]: Ensures the field contains a number less than or equal to the specified value.

3. Date Validation Rules
- valid_date[format]: Ensures the field contains a valid date, optionally checking against a format (e.g., valid_date[Y-m-d]).
- valid_date[Y-m-d]: Checks for a valid date in YYYY-MM-DD format.
- valid_date_format[format]: Checks that the date is in a specified format (e.g., Y-m-d for YYYY-MM-DD).
- String Validation Rules
- regex_match[pattern]: Ensures the field matches the specified regular expression pattern.
- valid_email: Ensures the field contains a valid email address.
- valid_emails: Ensures the field contains a comma-separated list of valid email addresses.
- valid_ip[ip_version]: Ensures the field contains a valid IP address, optionally specifying IPv4 or IPv6.
- valid_base64: Ensures the field contains a valid Base64-encoded string.
- valid_url: Ensures the field contains a valid URL.

5. File Upload Validation Rules

These rules are used to validate files when uploading.
- uploaded[file_field]: Ensures that a file was uploaded.
- max_size[file_field, size]: Ensures the uploaded file does not exceed a certain size (in kilobytes).
- max_dims[file_field, width, height]: Ensures the uploaded image file dimensions do not exceed the specified width and height.
- is_image[file_field]: Ensures the uploaded file is an image.
- mime_in[file_field, mime1, mime2, ...]: Ensures the uploaded file has one of the allowed MIME types (e.g., mime_in[userfile, image/jpg, image/png]).
- ext_in[file_field, ext1, ext2, ...]: Ensures the uploaded file has one of the specified extensions (e.g., ext_in[userfile, jpg, png]).

6. Custom Validation Rules
You can create custom validation rules in CodeIgniter by defining your own rule functions.

Define the Rule in a Controller or Custom Library:

```
$validation->setRule('field_name', 'label', 'custom_rule');
```

*Create the Custom Rule Method: Define the custom rule function, either in a custom library or directly within your controller if you're using it in a specific context.

Using CodeIgniter's built-in libraries (e.g., Database, Security, Validation)

Database Library

The Database library helps interact with your database, allowing you to perform queries and manipulate data.

1. Configure Database:
   Open app/Config/Database.php and add your database credentials:

```
public $default = [
   'DSN'     => '',
   'hostname' => 'localhost',
   'username' => 'your_username',
   'password' => 'your_password',
   'database' => 'your_database',
   'DBDriver' => 'MySQLi',
];
```

2. Using the Database in a Controller:
   In app/Controllers, create a new file named DatabaseController.php:

```
<?php
namespace App\Controllers;

use CodeIgniter\Controller;
use Config\Database;

class DatabaseController extends Controller
{
   public function getProducts()
   {
      $db = Database::connect();
      $query = $db->query("SELECT * FROM products");
      $data['products'] = $query->getResult();

      return view('products', $data);
   }
}
```

3. Create a View:
   In app/Views, create products.php to display the data:

```
<h1>Products</h1>
<?php foreach ($products as $product): ?>
   <p><?= esc($product->name) ?>: $<?= esc($product->price) ?></p>
<?php endforeach; ?>
```

4. Define Route:
   In app/Config/Routes.php, add a route:

```
$routes->get('products', 'DatabaseController::getProducts');
```

   *Access http://localhost:8080/products to view the list of products.

Security Library

The Security library provides functions for data sanitization and CSRF protection.

1. Escaping Output:
   Escaping prevents Cross-Site Scripting (XSS) attacks by sanitizing output.
   In app/Views/products.php, use:

```
<p><?= esc($product->name) ?></p>
```

2. CSRF Protection:
   Enable CSRF protection in app/Config/Filters.php by setting csrfProtection to true.
   Add CSRF protection to forms:

```
<form action="/submit" method="post">
  <?= csrf_field() ?>
  <input type="text" name="name" placeholder="Product Name">
  <button type="submit">Submit</button>
</form>
```

3. Sanitizing Data:
   Use to sanitize filenames:

```
$this->security->sanitizeFilename($filename).
```

Validation Library

The Validation library allows defining rules to validate form inputs.

1. Define Validation Rules:
   In app/Controllers, create a file called ValidationController.php:

```
<?php
namespace App\Controllers;

use CodeIgniter\Controller;

class ValidationController extends Controller
{
  public function submit()
  {
    $validation = \Config\Services::validation();

    $rules = [
      'name' => 'required|min_length[3]',
      'email' => 'required|valid_email'
    ];

    if (! $this->validate($rules)) {
      return view('form', [
        'validation' => $this->validator
      ]);
    }

    // Process valid data here
  }
}
```

2. Create a Form View:

   In app/Views, create form.php:

```
<form action="/submit" method="post">
  <input type="text" name="name" placeholder="Name">
  <input type="email" name="email" placeholder="Email">
  <button type="submit">Submit</button>

  <?php if (isset($validation)): ?>
    <div><?= $validation->listErrors() ?></div>
  <?php endif; ?>
</form>
```

3. Define Route:

```
$routes->post('submit', 'ValidationController::submit');
```

   *Access the form at http://localhost:8080/form.

Session Library

The Session library helps manage user sessions and store temporary data.

1. Start a Session:
   In app/Controllers, create SessionController.php:

```
<?php
namespace App\Controllers;

use CodeIgniter\Controller;

class SessionController extends Controller
{
  public function setSession()
  {
    session()->set('username', 'JohnDoe');
    return redirect()->to('/session-check');
  }

  public function checkSession()
  {
    $username = session()->get('username');
    return view('session', ['username' => $username]);
  }
}
```

2. Create a View:
   In app/Views, create session.php:

```
<h1>Welcome, <?= esc($username) ?></h1>
```

3. Define Routes:

```
$routes->get('session-set', 'SessionController::setSession');
$routes->get('session-check', 'SessionController::checkSession');
```

*Navigate to http://localhost:8080/session-set to set the session and then to http://localhost:8080/session-check to view it.

Email Library

The Email library helps send emails from within your application.

1. Configure Email:
   Open app/Config/Email.php and set up your email provider (SMTP or other):

```
public $protocol = 'smtp';
public $SMTPHost = 'smtp.example.com';
public $SMTPUser = 'your_email@example.com';
public $SMTPPass = 'your_password';
public $SMTPPort = 587;
```

2. Create Email Sending Controller:
   In app/Controllers, create EmailController.php:

```
<?php
namespace App\Controllers;

use CodeIgniter\Controller;

class EmailController extends Controller
{
   public function sendEmail()
   {
      $email = \Config\Services::email();

      $email->setFrom('your_email@example.com', 'Your Name');
      $email->setTo('recipient@example.com');
      $email->setSubject('Test Email');
      $email->setMessage('This is a test email from CodeIgniter.');

      if ($email->send()) {
         echo "Email sent successfully";
      } else {
         echo "Failed to send email";
         print_r($email->printDebugger(['headers']));
      }
   }
}
```

3. Define Route:

```
$routes->get('send-email', 'EmailController::sendEmail');
```

*Access http://localhost:8080/send-email to trigger the email.

Model-View-Controller (MVC) Pattern

Understanding the roles of models, views, and controllers

In CodeIgniter, Models, Views, and Controllers (MVC) each play a distinct role in separating the logic, data handling, and presentation of a web application.

Example 1: Setting Up a Basic MVC Structure

Create a Controller:

In app/Controllers, create ProductController.php:

```
<?php
namespace App\Controllers;

use App\Models\ProductModel;

class ProductController extends BaseController
{
   public function index()
   {
      $model = new ProductModel();
      $data['products'] = $model->findAll();
      return view('product/index', $data);
   }
}
```

Create a Model:

In app/Models, create ProductModel.php:

```
<?php
namespace App\Models;

use CodeIgniter\Model;

class ProductModel extends Model
{
   protected $table = 'products';
   protected $allowedFields = ['name', 'price', 'description'];
}
```

Create a View:

In app/Views/product, create index.php:

```
<h1>Product List</h1>
<?php foreach ($products as $product): ?>
   <p><?= esc($product['name']) ?>: $<?= esc($product['price']) ?></p>
<?php endforeach; ?>
```

Define a Route:

In app/Config/Routes.php, add:

```
$routes->get('products', 'ProductController::index');
```

*Access http://localhost:8080/products to see the product list.

Example 2: Adding a Create Method in the Controller

```
public function create()
{
   return view('product/create');
}
```

Example 3: Adding a Store Method in the Controller

```
public function store()
{
   $model = new ProductModel();
   $model->save($this->request->getPost());
   return redirect()->to('/products');
}
```

Creating database models and performing database operations

Models represent database tables and allow for database operations.

Example 1: Creating a Model for CRUD Operations

Create Product Model (if not created above):

```
protected $table = 'products';
```

Example 2: Fetching All Records

```
$products = $model->findAll();
```

Example 3: Fetching a Single Record by ID

Controller Method:

In ProductController, add a show method:

```
public function show($id)
{
   $model = new \App\Models\ProductModel();
   $data['product'] = $model->find($id);

   return view('product/show', $data);
}
```

Define a Route:

In app/Config/Routes.php, add:

```
$routes->get('product/(:num)', 'ProductController::show/$1');
```

Create a View:

In app/Views/product, create show.php:

```
<h1>Product Details</h1>
<p>Name: <?= esc($product['name']) ?></p>
<p>Price: $<?= esc($product['price']) ?></p>
<p>Description: <?= esc($product['description']) ?></p>
```

*Access http://localhost:8080/product/1 to view a specific product by ID.

Example 4: Updating a Record

Controller Method:

In ProductController, add edit and update methods:

```
public function edit($id)
{
   $model = new \App\Models\ProductModel();
   $data['product'] = $model->find($id);

   return view('product/edit', $data);
}

public function update($id)
{
   $model = new \App\Models\ProductModel();
   $model->update($id, $this->request->getPost());

   return redirect()->to('/products');
}
```

Define Routes:

In app/Config/Routes.php, add:

```
$routes->get('product/edit/(:num)', 'ProductController::edit/$1');
$routes->post('product/update/(:num)', 'ProductController::update/$1');
```

Create the Edit View:

In app/Views/product, create edit.php:

```
<h1>Edit Product</h1>
<form action="/product/update/<?= esc($product['id']) ?>" method="post">
   <input type="text" name="name" value="<?= esc($product['name']) ?>">
   <input type="text" name="price" value="<?= esc($product['price']) ?>">
   <textarea name="description"><?= esc($product['description']) ?></textarea>
   <button type="submit">Update Product</button>
</form>
```

Example 5: Deleting a Record

Controller Method:

In ProductController, add a delete method:

```
public function delete($id)
{
   $model = new \App\Models\ProductModel();
   $model->delete($id);

   return redirect()->to('/products');
}
```

Define Route:

In app/Config/Routes.php, add:

```
$routes->get('product/delete/(:num)', 'ProductController::delete/$1');
```

Using view templates and layouts

Using layouts allows us to define a base template and extend it in individual views.

Example 1: Creating a Layout Template

Create a Layout File: In app/Views/layouts, create main.php:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title><?= esc($title ?? 'My App') ?></title>
</head>
<body>
  <header>
    <h1>My App</h1>
  </header>
  <main>
    <?= $this->renderSection('content') ?>
  </main>
</body>
</html>
```

Example 2: Extending the Layout in Views

Use the Layout in a View: In app/Views/product/index.php, use:

```
<?= $this->extend('layouts/main') ?>

<?= $this->section('content') ?>
<h1>Product List</h1>
<!-- product list code here -->
<?= $this->endSection() ?>
```

Example 3: Passing Data to the Layout

In Controller: Pass a title to the view:

```
return view('product/index', ['title' => 'Products']);
```

Example 4: Adding a Navbar in Layout

Edit main.php Layout: Add a simple navbar:

```
<nav>
  <a href="/">Home</a> | <a href="/products">Products</a>
</nav>
```

Example 5: Adding a Footer Section

In main.php Layout: Add a footer:

```
<footer>
  <p>My App &copy; <?= date('Y') ?></p>
</footer>
```

Implementing custom filters and callbacks

Filters allow us to apply actions to requests, such as authentication or logging.

Example 1: Creating a Custom Filter

In app/Filters, create AuthFilter.php:

```
<?php
namespace App\Filters;

use CodeIgniter\HTTP\RequestInterface;
use CodeIgniter\HTTP\ResponseInterface;
use CodeIgniter\Filters\FilterInterface;

class AuthFilter implements FilterInterface
{
   public function before(RequestInterface $request, $arguments = null)
   {
      if (! session()->get('isLoggedIn')) {
         return redirect()->to('/login');
      }
   }

   public function after(RequestInterface $request, ResponseInterface $response, $arguments = null)
   {
      // Do something here
   }
}
```

Register the Filter:

Open app/Config/Filters.php and register it:

```
public $aliases = [
   'auth' => \App\Filters\AuthFilter::class,
];
```

Example 2: Applying the Filter to Routes

In app/Config/Routes.php, apply the filter to specific routes:

```
$routes->get('dashboard', 'DashboardController::index', ['filter' => 'auth']);
```

Example 3: Using Callbacks in Models

Create Callbacks for a Model: In ProductModel, define callbacks:

```
protected $beforeInsert = ['beforeInsert'];
protected $afterInsert = ['afterInsert'];

protected function beforeInsert(array $data)
{
   $data['data']['name'] = strtoupper($data['data']['name']);
   return $data;
}
```

```
protected function afterInsert(array $data)
{
   log_message('info', 'Product added with ID: ' . $data['id']);
}
```

Example 4: Creating a Custom Callback

Define Custom Callback: Add a custom callback to sanitize inputs:

```
protected function sanitizeInput(array $data)
{
   $data['data'] = array_map('htmlspecialchars', $data['data']);
   return $data;
}
```

Add to Callbacks:

```
protected $beforeInsert = ['sanitizeInput'];
```

Example 5: Logging Actions with Filters and Callbacks

Create Logging in Filter: Add log messages in AuthFilter:

```
public function before(RequestInterface $request, $arguments = null)
{
   log_message('info', 'User attempting to access: ' . current_url());
}
```

Authentication and Authorization

Implementing user authentication (e.g., password hashing, session management)

Example 1: Setting Up the User Database Table

Create a Users Table:

Use the following SQL to set up a basic users table with role and password_hash fields:

```
CREATE TABLE users (
   id INT AUTO_INCREMENT PRIMARY KEY,
   username VARCHAR(50) NOT NULL UNIQUE,
   password_hash VARCHAR(255) NOT NULL,
   role VARCHAR(20) NOT NULL
);
```

Example 2: Registering Users with Password Hashing

Create a Registration Form:

In app/Views/user/register.php, create a form:

```
<form action="/register" method="post">
   <input type="text" name="username" placeholder="Username" required>
   <input type="password" name="password" placeholder="Password" required>
   <button type="submit">Register</button>
</form>
```

Hash the Password and Store the User:

In app/Controllers/AuthController.php, create a method for registration:

```
public function register()
{
   $password = $this->request->getPost('password');
   $hashedPassword = password_hash($password, PASSWORD_BCRYPT);

   $data = [
      'username' => $this->request->getPost('username'),
      'password_hash' => $hashedPassword,
      'role' => 'user' // Default role
   ];

   $model = new \App\Models\UserModel();
   $model->insert($data);

   return redirect()->to('/login');
}
```

Example 3: Logging In with Password Verification
Create a Login Form:

In app/Views/user/login.php, create a form:

```
<form action="/login" method="post">
   <input type="text" name="username" placeholder="Username" required>
   <input type="password" name="password" placeholder="Password" required>
   <button type="submit">Login</button>
</form>
```

Verify the Password and Set the Session:
In AuthController, add a login method:

```
public function login()
{
   $username = $this->request->getPost('username');
   $password = $this->request->getPost('password');

   $model = new \App\Models\UserModel();
   $user = $model->where('username', $username)->first();

   if ($user && password_verify($password, $user['password_hash'])) {
      session()->set('isLoggedIn', true);
      session()->set('user_id', $user['id']);
      session()->set('role', $user['role']);
      return redirect()->to('/dashboard');
   } else {
      return redirect()->to('/login')->with('error', 'Invalid credentials');
   }
}
```

Example 4: Protecting Routes with Session Management
Create a Filter for Authentication:

In app/Filters, create AuthFilter.php:

```
<?php
namespace App\Filters;

use CodeIgniter\HTTP\RequestInterface;
use CodeIgniter\HTTP\ResponseInterface;
use CodeIgniter\Filters\FilterInterface;

class AuthFilter implements FilterInterface
{
   public function before(RequestInterface $request, $arguments = null)
   {
      if (! session()->get('isLoggedIn')) {
         return redirect()->to('/login');
      }
   }

   public function after(RequestInterface $request, ResponseInterface $response, $arguments = null)
   {
      // Do nothing
   }
}
```

Register and Apply the Filter:

In app/Config/Filters.php, add the filter:

```
public $aliases = [
   'auth' => \App\Filters\AuthFilter::class,
];
```

Protect routes in app/Config/Routes.php:

```
$routes->get('dashboard', 'DashboardController::index', ['filter' => 'auth']);
```

Example 5: Logging Out and Clearing the Session
Create a Logout Method:

In AuthController, add:

```
public function logout()
{
   session()->destroy();
   return redirect()->to('/login');
}
```

Add a Logout Link:
In your main layout or navigation, add a link to /logout.

Implementing role-based access control (RBAC)

RBAC allows restricting access based on user roles (e.g., admin, user).

Example 1: Adding a Role Field in the Users Table

Ensure the users table has a role column (e.g., user or admin).

Example 2: Adding Role-Based Restrictions in Controllers

Check User Role in Controller:

In DashboardController, add a method to restrict access:

```
public function adminDashboard()
{
   if (session()->get('role') !== 'admin') {
      return redirect()->to('/dashboard')->with('error', 'Access Denied');
   }

   // Load admin-specific view
   return view('admin/dashboard');
}
```

Define Route for Admin Dashboard:

Add in Routes.php:

```
$routes->get('admin/dashboard', 'DashboardController::adminDashboard', ['filter' => 'auth']);
```

Example 3: Role-Based Menu Links in Views

Show Links Based on Role:

In the main navigation, conditionally display links:

```
<?php if (session()->get('role') === 'admin'): ?>
   <a href="/admin/dashboard">Admin Dashboard</a>
<?php endif; ?>
```

Example 4: Role-Based Access Control Filter

Create a Custom Filter for Role Access:

In app/Filters, create AdminFilter.php:

```
public function before(RequestInterface $request, $arguments = null)
{
   if (session()->get('role') !== 'admin') {
      return redirect()->to('/dashboard')->with('error', 'Access Denied');
   }
}
```

Example 5: Applying Role-Based Filters in Routes

Apply the Filter in Routes:

Protect routes using admin filter:

```
$routes->get('admin-only', 'AdminController::index', ['filter' => 'admin']);
```

Using Single-Sign On (SSO)

SSO allows users to authenticate once and gain access to multiple applications. Here we'll use OAuth2 as a sample SSO.

Example 1: Install OAuth2 Client Library

Install OAuth2 Client Library:

```
composer require league/oauth2-client
```

Example 2: Configure OAuth2 Provider

Create OAuth2 Provider Configuration:

In app/Config/OAuth.php:

```
public $providers = [
   'google' => [
      'clientId'     => 'YOUR_GOOGLE_CLIENT_ID',
      'clientSecret' => 'YOUR_GOOGLE_CLIENT_SECRET',
      'redirectUri'  => 'YOUR_REDIRECT_URI',
      'hostedDomain' => 'YOUR_DOMAIN.com',
   ]
];
```

Example 3: Redirecting User to OAuth Provider

Controller Method:

In AuthController, add:

```
public function redirectToProvider()
{
   $provider = new \League\OAuth2\Client\Provider\Google($config->providers['google']);
   $authUrl = $provider->getAuthorizationUrl();
   session()->set('oauth2state', $provider->getState());
   return redirect()->to($authUrl);
}
```

Example 4: Handling OAuth2 Callback

OAuth Callback Handling:

Capture the token and user details in AuthController:

```
public function handleProviderCallback()
{
   $provider = new \League\OAuth2\Client\Provider\Google($config->providers['google']);
   $token = $provider->getAccessToken('authorization_code', ['code' => $_GET['code']]);
   $user = $provider->getResourceOwner($token);
   session()->set('user', $user->toArray());
   return redirect()->to('/dashboard');
}
```

Example 5: Displaying User Information Post-Login

Display User Data in Dashboard:

In the dashboard view, show user info:

```
<?php $user = session()->get('user'); ?>
<p>Welcome, <?= esc($user['name']) ?></p>
```

Introduction to Docker and Containerization

Introduction to containers and containerization

Containers are lightweight, standalone packages that include an application and all its dependencies, ensuring that the application runs consistently in any environment.

Key Concepts

- ✓ Isolation: Containers run independently from each other, allowing multiple applications to run on the same system without conflict.
- ✓ Portability: Containerized applications can run on any machine that supports container runtime, like Docker.
- ✓ Efficiency: Containers share the host OS kernel, which makes them lightweight compared to virtual machines.

Benefits of using Docker for development and deployment

Using Docker for development and deployment offers several advantages:

- ✓ Consistency: Docker ensures that the application runs the same way across different environments (local, staging, production).
- ✓ Scalability: Docker containers can be easily scaled up or down to handle varying loads.
- ✓ Resource Efficiency: Containers are lightweight and consume fewer resources compared to virtual machines.
- ✓ Rapid Deployment: Docker simplifies the process of setting up and deploying applications.

Docker architecture and components

Docker consists of several key components:

- Docker Engine: The core of Docker, responsible for creating, running, and managing containers.
- Docker Images: The templates that contain all necessary files and dependencies for a container.
- Docker Containers: Instances of Docker images that run applications in isolated environments.
- Dockerfile: A file with instructions to build Docker images.
- Docker Hub: A cloud-based repository where Docker images can be stored and shared.

Installing Docker on Windows

1. Download Docker Desktop:

Go to Docker Desktop for Windows and download the installer
https://docs.docker.com/desktop/install/windows-install/

2. Install Docker Desktop:

Run the downloaded installer and follow the installation wizard.
Select the option to enable WSL 2 integration (if available). Windows Subsystem for Linux (WSL 2) allows Docker to run in a Linux environment on Windows.

3. Start Docker Desktop:
   Launch Docker Desktop from the Start Menu.

4. Verify Installation:
   Open Command Prompt or PowerShell and type:

```
docker --version
```

   You should see the Docker version installed on your machine.

5. Run a Test Container:
   Run a simple container to verify Docker is working:

```
docker run hello-world
```

   *This will pull the hello-world image from Docker Hub and run it, displaying a message confirming Docker is working.

Creating Docker images

Install CodeIgniter 4 Locally:

1. Create a new CodeIgniter 4 application locally.
   Open your Command Prompt or PowerShell and navigate to the directory where you want to create the project:

```
composer create-project codeigniter4/appstarter my-codeigniter-app
```

   This will create a new folder named my-codeigniter-app with all CodeIgniter files.

2. Test the Application Locally:
   Enter the project directory and start the local server:

```
cd my-codeigniter-app
php spark serve
```

   *Go to http://localhost:8080 in your browser to ensure that the application is running.

A Dockerfile is a script containing commands to build a Docker image.

1. Create a Dockerfile:
   In the root directory of your CodeIgniter project (my-codeigniter-app), create a file named Dockerfile (without any file extension).

2. Write Dockerfile Instructions:
   Open the Dockerfile and add the following instructions:

```
# Use an official PHP image with Apache
FROM php:8.0-apache

# Install required PHP extensions
RUN docker-php-ext-install mysqli pdo pdo_mysql

# Set the working directory
WORKDIR /var/www/html

# Copy the current directory contents into the container
COPY . /var/www/html

# Set permissions for the storage and writable directories
RUN chown -R www-data:www-data /var/www/html/writable /var/www/html/cache
RUN chmod -R 775 /var/www/html/writable /var/www/html/cache

# Expose port 80
EXPOSE 80
```

Explanation of Dockerfile Instructions:

FROM php:8.0-apache	:Starts with a PHP image that has Apache installed.
RUN docker-php-ext-install mysqli pdo pdo_mysql	:Installs PHP extensions needed for CodeIgniter.
WORKDIR /var/www/html	:Sets the working directory in the container.
COPY . /var/www/html	:Copies all application files to the container's /var/www/html directory.
RUN chown... and chmod...	:Sets permissions for writable directories to avoid permission issues.
EXPOSE 80	:Exposes port 80 to access the application.

3. Create a docker-compose.yml File (Optional)

Using Docker Compose can simplify the setup, especially if you plan to add more services like a database.

Create a docker-compose.yml file:
In the project root (my-codeigniter-app), create a file named docker-compose.yml.

Add the Docker Compose Configuration:

```
version: '3.8'

services:
 app:
  build: .
  ports:
   - "8080:80"
  volumes:
   - .:/var/www/html
  environment:
   - CI_ENVIRONMENT=development
 db:
  image: mysql:8.0
  environment:
   MYSQL_ROOT_PASSWORD: rootpassword
   MYSQL_DATABASE: codeigniter
   MYSQL_USER: user
   MYSQL_PASSWORD: password
  ports:
   - "3306:3306"
```

Explanation of docker-compose.yml:

App	: Defines the CodeIgniter application service.
Build	: Builds the image using the Dockerfile in the current directory.
Ports	: Maps port 8080 on the host to port 80 in the container.
Volumes	: Maps the current directory to /var/www/html in the container, allowing live updates.
Db	: Sets up a MySQL database service with a default database and user.

4. Build the Docker Image

Open PowerShell or Command Prompt, navigate to the my-codeigniter-app directory.

```
docker-compose build
```

*This command reads the Dockerfile and docker-compose.yml, and builds the application and database images.

Running containers

1. Use Docker Compose to start both the application and the database containers:

```
docker-compose up
```

*You should see output indicating that the containers are running.

2. Open a web browser and go to http://localhost:8080. You should see your CodeIgniter application running inside the Docker container.

Managing containers and images

Here are some useful commands for managing your Docker containers and images.

Stop the Containers:

```
docker-compose down
```

*This stops and removes the containers defined in docker-compose.yml.

View Running Containers:

```
docker ps
```

*Shows all currently running containers.

Remove Unused Images:

```
docker image prune
```

*Removes unused Docker images to free up disk space.

Restart Containers:

```
docker-compose up -d
```

*Adds -d to run containers in detached mode (in the background).

Check Logs:

```
docker-compose logs
```

*Displays logs for all services in docker-compose.yml.

Troubleshooting Tips

- Permission Issues: If you encounter permission issues, double-check the permissions set for writable and cache directories in the Dockerfile.
- If you see “Whoops….We encountered a snag”, your environment is still set to production. Make sure you have a (.env) file (not just the (env) file and it should be set to use “development” not “production” (set as default).
- Database Connection: Update CodeIgniter’s .env file with database connection settings to use the db service (as defined in docker-compose.yml).

Docker vs Docker-Compose

1. **docker** Command

The docker command is the primary CLI tool for interacting with Docker. It's mainly used to manage individual containers, images, networks, and volumes. docker commands are lower-level and more granular, focusing on single containers or images at a time.

Common docker Commands

Images:

docker build -t image_name . – Build an image from a Dockerfile.
docker pull image_name – Download an image from Docker Hub or a repository.
docker images – List all local images.

Containers:

docker run -d -p host_port:container_port image_name – Run a container.
docker ps – List running containers.
docker stop container_id – Stop a container.
docker rm container_id – Remove a stopped container.

Networks and Volumes:

docker network ls – List Docker networks.
docker volume create volume_name – Create a Docker volume.

2. **docker-compose** Command

The docker-compose command is a higher-level tool used to define and manage multi-container Docker applications. It reads configurations from a docker-compose.yml file, where you can specify multiple services (containers), networks, and volumes in a single configuration.

Common docker-compose Commands

Starting Services:

docker-compose up – Build, (re)create, start, and attach to containers for all services defined in the docker-compose.yml file.
docker-compose up -d – Start containers in detached mode (background).
docker-compose build – Build or rebuild services.

Stopping Services:

docker-compose down – Stop and remove containers, networks, and volumes created by up.
docker-compose stop – Stop running containers without removing them.

Viewing Status:

docker-compose ps – List containers related to the docker-compose.yml file.

Logs and Debugging:

docker-compose logs – View output from services.
docker-compose exec service_name command – Run a command inside a specific service container.

Containerizing CodeIgniter 4 Applications with Docker

Defining the base image for your CodeIgniter 4 application

The base image is the starting point for building your Docker image. Since CodeIgniter is a PHP application, we'll use an official PHP image with Apache.

1. Create a Dockerfile:

Inside your CodeIgniter project root directory (e.g., my-codeigniter-app), create a file named Dockerfile (no file extension).

2. Set the Base Image in the Dockerfile:

Add the following line to define the base image:

dockerfile

```
# Use PHP 8.1 as the base image
FROM php:8.1-apache
```

Note(s):

- ✓ php:8.1-apache is an official PHP image with Apache pre-installed. CodeIgniter 4.5.x requires PHP 8.1+.

Copying application files

Copy the CodeIgniter application files into the Docker image.

1. Set the Working Directory:

Set the directory inside the container where the files will be copied:

dockerfile

```
# Set the working directory
WORKDIR /var/www/html
```

2. Copy Files:

Add a command to copy all project files into the container:

dockerfile

```
# Copy application files to the working directory
COPY . /var/www/html
```

Note(s):

- ✓ WORKDIR sets the directory within the container where all following commands will run.
- ✓ COPY . /var/www/html copies all files from the current directory (CodeIgniter app) to /var/www/html in the container, which is the default location for Apache's root directory.

Installing dependencies

CodeIgniter might require PHP extensions, such as pdo_mysql for MySQL. We can install these in the Dockerfile.

1. Install Dependencies:

Add the following line to install any necessary PHP extensions:

dockerfile

```
# Install necessary PHP extensions, including intl
RUN apt-get update && \
   apt-get install -y libicu-dev && \
   docker-php-ext-install intl pdo pdo_mysql

# Enable Apache mod_rewrite for CodeIgniter
RUN a2enmod rewrite
```

Set Permissions

Set permissions for the folders and files that will be stored in the docker image

```
# Set the necessary permissions
RUN chown -R www-data:www-data /var/www/html \
   && chmod -R 755 /var/www/html
```

Exposing ports

To make the application accessible, we need to expose a port on which the container will listen.

1. Expose Port 80:

dockerfile

```
# Expose port 80 for Apache
EXPOSE 80
```

Setting environment variables

Environment variables allow us to configure our application's behavior. You can define variables within the Dockerfile or in a docker-compose.yml file.

1. Set Environment Variable for Development:

dockerfile

```
# Set the environment for CodeIgniter
ENV CI_ENVIRONMENT=development
```

Note(s):

- ✓ ENV defines environment variables inside the Docker container. Here, we're setting CI_ENVIRONMENT to development for debugging and testing purposes in CodeIgniter.

Create Environment File

Copy ***env*** file to ***.env***

.env

```
...
CI_ENVIRONMENT = development
...
```

Create .dockerignore

Create .dockerignore in project root. This file indicates the other files that will not be included in docker.

.dockerignore

```
.git
vendor
.env
.gitignore
```

Create docker-compose.yml

This file contains configuration for the docker container

docker-compose.yml

```
version: '3.8'

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: codeigniter-app
    ports:
      - '8080:80'
    volumes:
      - .:/var/www/html
    environment:
      - APP_ENV=local
      - APP_DEBUG=true
```

Building and Running a CodeIgniter 4 Docker Image

Let's build the image and run it as a container.

1. Build the Docker Image:

Open Command Prompt or PowerShell in your project directory (my-codeigniter-app) and run the following command:

```
docker build -t codeigniter-app .
# or
docker-compose build
```

*This command tells Docker to build an image from the Dockerfile in the current directory (.) and tag it as codeigniter-app.

2. Run the Docker Container:

Run the container, mapping port 8080 on your host to port 80 in the container:

```
docker-compose up -d
# or
docker run -d -p 8080:80 --name my-codeigniter-app codeigniter-app
```

-d runs the container in detached mode (in the background).
-p 8080:80 maps port 80 in the container to port 8080 on the host machine.
--name my-codeigniter-app names the container instance my-codeigniter-app.

3. Verify the Container is Running:

To check if the container is running, use:

```
docker ps
```

*You should see my-codeigniter-app listed with port 8080 mapped.

4. Access the Application:

Open a web browser and navigate to http://localhost:8080. You should see your CodeIgniter 4 application running inside the Docker container.

5. Stopping and Removing the Container (Optional):

To stop the container, use:

```
docker stop my-codeigniter-app
```

To remove the container:

```
docker rm my-codeigniter-app
```

Full Dockerfile for CodeIgniter 4

Here's the complete Dockerfile after all steps:

dockerfile

```
# Use PHP 8.1 as the base image
FROM php:8.1-apache

# Install necessary PHP extensions, including intl
RUN apt-get update && \
   apt-get install -y libicu-dev && \
   docker-php-ext-install intl pdo pdo_mysql

# Enable Apache mod_rewrite for CodeIgniter
RUN a2enmod rewrite

# Copy the app files to the working directory
COPY . /var/www/html

# Set the working directory
WORKDIR /var/www/html

# Set the necessary permissions
RUN chown -R www-data:www-data /var/www/html \
   && chmod -R 755 /var/www/html

# Expose the necessary port
EXPOSE 80
```