CodeIgniter4 Additional Topics

Model Class Built-in Functions

1. find()

Usage: Use find() to get one or more specific records.

```
$model->find(1);         // Finds the record with primary key 1
$model->find([1, 2, 3]);   // Finds records with primary keys 1, 2, and 3
$model->findAll();       // Retrieves all records
```

2. findAll()

Usage: Use findAll() when you want all records, with optional limit and offset.

```
$model->findAll();          // Retrieves all records
$model->findAll(10, 20);     // Retrieves 10 records starting from the 20th record
```

3. first()

Usage: Use first() to get the first matching row in a query.

```
$model->where('email', 'john@example.com')->first();
```

4. delete()

Usage: Use delete() to remove records.

```
$model->delete(1);          // Deletes the record with primary key 1
$model->where('status', 'inactive')->delete(); // Deletes records where status is inactive
```

5. truncate()

Usage: Use truncate() when you want to clear the table completely.

```
$model->truncate();
```

6. countAll()

Usage: Use countAll() to get a count of all entries.

```
$totalRecords = $model->countAll();
```

7. countAllResults()

Usage: Use countAllResults() for filtered counts with specific conditions.

```
$totalActiveUsers = $model->where('status', 'active')->countAllResults();
```

8. select()

Usage: Use select() to customize the columns returned in a query.

```
$model->select('name, email')->findAll();
```

9. asArray() and asObject()

Usage: Use asArray() or asObject() to control result format.

```
$model->asArray()->findAll();  // Returns results as arrays
$model->asObject()->findAll();  // Returns results as objects
```

10. set()
Usage: Use set() to define fields before calling another method.

```
$model->set('name', 'Jane Doe')->update(1);   // Updates the name field for the record with primary key 1
```

11. where() and orWhere()
Usage: Use where() to filter records based on specific conditions.

```
$model->where('status', 'active')->findAll();
```

12. like() and orLike()
Usage: Use like() for partial text matching.

```
$model->like('name', 'Doe')->findAll();   // Finds records with 'Doe' in the name
```

13. orderBy()
Usage: Use orderBy() to define sorting in ascending or descending order.

```
$model->orderBy('created_at', 'DESC')->findAll();
```

14. limit() and offset()
Usage: Use limit() and offset() to control pagination or subset retrieval.

```
$model->limit(10, 20)->findAll();   // Retrieves 10 records starting from the 20th
```

Using Raw SQL

1. Setting Up the Database Connection
Before you start, make sure to load the database in your controller or model:

```
$db = \Config\Database::connect();
```

2. Create (Insert) Operation
To insert a record using raw SQL:

```
$sql = "INSERT INTO users (name, email) VALUES (:name:, :email:)";
$db->query($sql, [
   'name'  => 'John Doe',
   'email' => 'john@example.com',
]);
```

Here, :name: and :email: are placeholders for binding parameters, which helps prevent SQL injection.

3. Read (Select) Operation
To retrieve records from the database using raw SQL:

Example 1: Fetch All Records

```
$sql = "SELECT * FROM users";
$query = $db->query($sql);
$results = $query->getResult(); // Retrieves results as objects
// $results = $query->getResultArray(); // To retrieve results as an associative array
```

Example 2: Fetch a Single Record with a Condition

```
$sql = "SELECT * FROM users WHERE id = :id:";
$query = $db->query($sql, ['id' => 1]);
$result = $query->getRow(); // Retrieves a single row as an object
// $result = $query->getRowArray(); // To retrieve as an associative array
```

4. Update Operation

To update a record using raw SQL:

```
$sql = "UPDATE users SET name = :name:, email = :email: WHERE id = :id:";
$db->query($sql, [
  'name'  => 'Jane Doe',
  'email' => 'jane@example.com',
  'id'    => 1,
]);
```

In this example, the user with id 1 will have their name and email fields updated.

5. Delete Operation

To delete a record using raw SQL:

```
$sql = "DELETE FROM users WHERE id = :id:";
$db->query($sql, ['id' => 1]);
```

This command will delete the record with id 1 from the users table.

6. Additional Tips for Using Raw SQL in CodeIgniter 4

Get Affected Rows: You can check the number of rows affected by an insert, update, or delete operation using:

```
$affectedRows = $db->affectedRows();
```

Query Builder Alternative: For many cases, CodeIgniter's Query Builder offers a cleaner, more secure way to handle SQL operations. However, raw SQL is valuable for complex joins, subqueries, or unsupported SQL features.

Error Handling: Check for SQL errors by capturing the last error message:

```
if (!$db->query($sql)) {
  $error = $db->error();
  // Handle error as needed
}
```

Microservices and API Gateway

To create a basic microservices architecture using CodeIgniter 4, we'll set up two separate services and a third service acting as an API gateway to communicate with them.

- User Service: Manages user data (e.g., creating and fetching user details).
- Order Service: Manages orders (e.g., creating and fetching orders for a user).
- API Gateway: Acts as a single entry point for clients, forwarding requests to the respective microservices.

Setting Up the Services

For simplicity, let's assume each service is a separate CodeIgniter 4 application with its own database.

1. User Service

Functionality: Handles CRUD operations for users.

User Table (Database Schema)

```
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100)
);
```

UserServiceController.php

In the UserServiceController, we'll create endpoints for creating and fetching users.

```
namespace App\Controllers;

use App\Models\UserModel;
use CodeIgniter\RESTful\ResourceController;

class UserServiceController extends ResourceController
{
  protected $modelName = 'App\Models\UserModel';
  protected $format    = 'json';

  public function createUser()
  {
    $data = [
      'name' => $this->request->getPost('name'),
      'email' => $this->request->getPost('email'),
    ];

    if ($this->model->insert($data)) {
      return $this->respondCreated(['status' => 'User created successfully']);
    } else {
      return $this->failValidationErrors($this->model->errors());
    }
  }

  public function getUser($id = null)
  {
    $user = $this->model->find($id);
    if ($user) {
      return $this->respond($user);
    } else {
      return $this->failNotFound('User not found');
```

```
    }
  }
}
```

UserModel.php

The model for interacting with the users table.

```
namespace App\Models;

use CodeIgniter\Model;

class UserModel extends Model
{
  protected $table     = 'users';
  protected $primaryKey = 'id';
  protected $allowedFields = ['name', 'email'];
}
```

2. Order Service

Functionality: Manages orders for users.

Order Table (Database Schema)

```
CREATE TABLE orders (
  id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT,
  product VARCHAR(100),
  amount DECIMAL(10, 2)
);
```

OrderServiceController.php

In the OrderServiceController, we'll create endpoints for creating and fetching orders.

```
namespace App\Controllers;

use App\Models\OrderModel;
use CodeIgniter\RESTful\ResourceController;

class OrderServiceController extends ResourceController
{
  protected $modelName = 'App\Models\OrderModel';
  protected $format    = 'json';

  public function createOrder()
  {
    $data = [
      'user_id' => $this->request->getPost('user_id'),
      'product' => $this->request->getPost('product'),
```

```
            'amount' => $this->request->getPost('amount'),
        ];

        if ($this->model->insert($data)) {
            return $this->respondCreated(['status' => 'Order created successfully']);
        } else {
            return $this->failValidationErrors($this->model->errors());
        }
    }

    public function getOrder($id = null)
    {
        $order = $this->model->find($id);
        if ($order) {
            return $this->respond($order);
        } else {
            return $this->failNotFound('Order not found');
        }
    }
}
```

OrderModel.php
The model for interacting with the orders table.

```
namespace App\Models;

use CodeIgniter\Model;

class OrderModel extends Model
{
    protected $table      = 'orders';
    protected $primaryKey = 'id';
    protected $allowedFields = ['user_id', 'product', 'amount'];
}
```

3. API Gateway
The API Gateway acts as a central entry point that forwards client requests to the User Service or Order Service.

ApiGatewayController.php

```
namespace App\Controllers;

use CodeIgniter\HTTP\ResponseInterface;
use CodeIgniter\API\ResponseTrait;

class ApiGatewayController extends BaseController
```

```
{
    use ResponseTrait;

    public function createUser()
    {
        $response = $this->forwardRequest('http://localhost:8081/user/create');
        return $this->respond($response->getBody(), $response->getStatusCode());
    }

    public function createOrder()
    {
        $response = $this->forwardRequest('http://localhost:8082/order/create');
        return $this->respond($response->getBody(), $response->getStatusCode());
    }

    public function getUser($id)
    {
        // string response that can be parsed:
        // $response = $this->forwardRequest("http://localhost:8081/user/{$id}");
        // return $this->respond($response->getBody(), $response->getStatusCode());

        $response = $this->forwardRequest("http://localhost:8081/user/{$id}");

        // Decode the response body to ensure it's treated as JSON
        $responseData = json_decode($response->getBody(), true);

        // Return as a JSON response with the appropriate HTTP status code
        return $this->respond($responseData, $response->getStatusCode());
    }

    public function getOrder($id)
    {
        // string response that can be parsed
        // $response = $this->forwardRequest("http://localhost:8082/order/{$id}");
        // return $this->respond($response->getBody(), $response->getStatusCode());

        $response = $this->forwardRequest("http://localhost:8082/order/{$id}");

        // Decode the response body to ensure it's treated as JSON
        $responseData = json_decode($response->getBody(), true);

        // Return as a JSON response with the appropriate HTTP status code
        return $this->respond($responseData, $response->getStatusCode());
    }
```

```
  private function forwardRequest($url)
  {
    $client = \Config\Services::curlrequest();
    return $client->request(
      $this->request->getMethod(),
      $url,
      [
        'headers' => $this->request->getHeaders(),
        'form_params' => $this->request->getPost(),
        'http_errors' => false
      ]
    );
  }
}
```

Setting Up Routes

Each service should have routes for its endpoints.

User Service Routes (UserService/app/Config/Routes.php)

```
$routes->post('user/create', 'UserServiceController::createUser');
$routes->get('user/(:num)', 'UserServiceController::getUser/$1');
```

Order Service Routes (OrderService/app/Config/Routes.php)

```
$routes->post('order/create', 'OrderServiceController::createOrder');
$routes->get('order/(:num)', 'OrderServiceController::getOrder/$1');
```

API Gateway Routes (ApiGateway/app/Config/Routes.php)

```
$routes->post('api/user/create', 'ApiGatewayController::createUser');
$routes->post('api/order/create', 'ApiGatewayController::createOrder');
$routes->get('api/user/(:num)', 'ApiGatewayController::getUser/$1');
$routes->get('api/order/(:num)', 'ApiGatewayController::getOrder/$1');
```

Running the Microservices

- ✓ Run each CodeIgniter application on different ports (e.g., User Service on localhost:8081, Order Service on localhost:8082, and API Gateway on localhost:8080).
- ✓ The client communicates with the API Gateway (e.g., localhost:8080/api/user/create), which forwards requests to the appropriate microservice.

Whitelisting IPs for API Access

1. Create a new filter file in the app/Filters directory (e.g., IpWhitelist.php).

```
<?php
namespace App\Filters;

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

class IpWhitelist implements FilterInterface
{
  public function before(RequestInterface $request, $arguments = null)
  {
    // Define the list of allowed IP addresses
    $whitelistedIps = [
      '192.168.1.10',  // example IP
      '203.0.113.15',  // another example IP
      '127.0.0.1',     // localhost (useful for testing)
      '::1'
    ];

    // Get the IP address of the client
    $clientIp = $request->getIPAddress();  // test this in a controller action to see your ip

    // Check if the client's IP is in the whitelist
    if (!in_array($clientIp, $whitelistedIps)) {
      // If not, return a forbidden response
      return \Config\Services::response()->setStatusCode(403)->setBody('Access denied');
    }

    // Allow the request to proceed if the IP is whitelisted
  }

  public function after(RequestInterface $request, ResponseInterface $response, $arguments = null)
  {
    // No action needed after the request
  }
}
```

2. Register the Filter in app/Config/Filters.php

Open the Filters.php file and add your custom filter to the aliases array:

```
public $aliases = [
  'ipWhitelist' => \App\Filters\IpWhitelist::class,
  // other filters
];
```

3. Apply the Filter to Your API Routes

You can apply the ipWhitelist filter to specific routes in app/Config/Routes.php:

```
$routes->group('api', ['filter' => 'ipWhitelist'], function($routes) {
  $routes->get('data', 'ApiController::getData');
  $routes->post('data', 'ApiController::postData');
  // Add other routes within this API group
});
```

4. Test the IP Whitelisting

When accessing the API endpoints, only requests from IPs listed in $whitelistedIps in the IpWhitelist filter will be allowed. If the request comes from an unapproved IP, the response will be a 403 Forbidden status with the message “Access denied.”